REPORT DOCUMENTATION PAGE

AFRL-SR-BL-TR-00-

0490

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	5/31/00	Final Technical (03/1/97-05/31/00)

4. TITLE AND SUBTITLE
Super-Resolution Processing and Fusion of Multisensor Data for Advanced Target Surveillance and Tracking

5. FUNDING NUMBERS
F49620-97-1-0243

6. AUTHOR(S)
Dr. Malur K. Sundareshan

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Department of Electrical and Computer Engineering
The College of Engineering and Mines
The University of Arizona
Tucson, AZ 85721

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)
AFOSR/NM
801 N. Randolph St Room 732
Arlington, VA 22203-1977

10. SPONSORING / MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION / AVAILABILITY STATEMENT
Approved for public release - Distribution is Unlimited

12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

Design of advanced surveillance and tracking systems typically employ multiple sensors which can provide large amounts of useful data to detect, identify and track targets of interest. Attempts at intelligent utilization of data for optimizing the processing efficiency in such multi-sensor operations require novel processing methods which need to be carefully tailored due to the disparate forms of data and the disparity in the resolution achievable from these sensors. This project was primarily aimed at the super-resolution processing of imagery data to improve the resolution in acquired images so that any problems arising from the disparity in resolution levels can be mitigated and efficient synthesis of fusion mechanisms can be developed. Specific investigations that were conducted as part of this project included: (i) A detailed analysis of resolution challenges addressing several important questions such as how to quantify resolution in an image (acquired or processed), (ii) development of systematic digital processing algorithms obtained by employing a statistical optimization framework for achieving resolution enhancement and super-resolution, and (iii) performance evaluation studies that included results of processing both simulated image data and results of processing PMMW data acquired from the radiometers being built by the Air Force Wright Laboratory Armament Directorate. The major accomplishments and research advances made in this project through rigorous mathematical analysis and extensive simulation experiments are outlined in this report.

20001002 018

14. SUBJECT TERMS

15. NUMBER OF PAGES
196

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UNLIMITED

SUPER-RESOLUTION PROCESSING AND FUSION OF MULTISENSOR DATA FOR ADVANCED TARGET SURVEILLANCE AND TRACKING

FINAL REPORT

(For AFOSR Grant # F49620-97-1-0243)

Prepared by: **Dr. Malur K. Sundareshan**
Professor of Electrical and Computer Engineering
University of Arizona
Tucson, AZ 85721-0104
Tel: (520) 621-2953; Fax: (520) 626-3144
e-mail: *sundareshan@ece.arizona.edu*

Prepared for: **Air Force Office of Scientific Research**
and
AF Wright Laboratory Armament Directorate

Program Manager: **Dr. Jon Sjogren**
AFOSR/NM, Arlington, VA 22203

May 31, 2000

SUPER-RESOLUTION PROCESSING AND FUSION OF MULTISENSOR DATA FOR ADVANCED TARGET SURVEILLANCE AND TRACKING

FINAL REPORT

Dr. Malur K. Sundareshan
Professor of Electrical and Computer Engineering
University of Arizona, Tucson, AZ 85721-0104

ABSTRACT

Design of advanced surveillance and tracking systems typically employ multiple sensors which can provide large amounts of useful data to detect, identify and track targets of interest. Attempts at intelligent utilization of data for optimizing the processing efficiency in such multi-sensor operations require novel processing methods which need to be carefully tailored due to the disparate forms of data and the disparity in the resolution achievable from these sensors. This project, which was sponsored by the Air Force Office of Scientific Research, was primarily aimed at the super-resolution processing of imagery data to improve the resolution in acquired images so that any problems arising from the disparity in resolution levels can be mitigated and efficient synthesis of fusion mechanisms can be developed. The principal objectives of this effort were to develop and evaluate specific techniques for the restoration and super-resolution of images collected from practical sensing operations. In order to increase the relevance of this work to the U.S. Air Force operations, a close liaison was maintained with the Air Force Wright Laboratory Armament Directorate. For proof of concept demonstrations, the tailoring of the algorithms as well as the processing studies were conducted with a special focus on achieving resolution enhancements in the passive millimeter-wave (PMMW) images acquired by the Air Force Wright Laboratory Armament Directorate with the state-of-the-art radiometers being built by them.

Specific investigations that were conducted as part of this project included: (i) A detailed analysis of resolution challenges addressing several important questions such as how to quantify resolution in an image (acquired or processed), (ii) development of systematic digital processing algorithms obtained by employing a statistical optimization framework for achieving resolution enhancement and super-resolution, and (iii) performance evaluation studies that included results of processing both simulated image data and results of processing PMMW data acquired from the radiometers being built by the Air Force Wright Laboratory Armament Directorate. The major accomplishments and research advances made in this project through rigorous mathematical analysis and extensive simulation experiments are outlined in this report.

TABLE OF CONTENTS

LIST OF FIGURES

1. INTRODUCTION

1.1 Principal Objectives of the Project

A variety of sensing devices ranging from radar systems to lasers and optical imaging systems are presently being developed for facilitating efficient target surveillance and tracking operations. The limitations of using a single sensor in these operations, such as limited accuracy and resolution and lack of robustness, have motivated the design of surveillance and tracking systems with multiple sensors which can provide large amounts of useful data to detect, track and identify targets of interest. However, current surveillance and tracking algorithms usually use information from only one sensor (such as a track-while-scan (TWS) radar) or attempt to combine information from different sensors in an *ad hoc* manner. While it is intuitive that using additional data available can result in improved detection, classification and track maintenance performance, attempting to include this data efficiently will require novel processing methods which need to be carefully tailored due to the disparate forms of data available from these sensors and the disparity in the resolution of the various sensors. While the problems associated the former (*viz.*, disparity of data forms) can be overcome by the development of appropriate feature-level fusion algorithms that are based on extracting a set of key features from each incoming data stream which is then followed by an integration of features abstracted from the various data streams, problems associated with the latter (*viz.*, disparity in resolution levels), particularly when imagery data is involved, often complicate the process of feature extraction and establishing correspondence (or registration) of images. Consequently improvement of the resolution in the acquired image data will almost always be necessary before this data can be further used to perform any decision-making either independently or by fusion with other forms of data.

The primary goal of this project was the development of advanced image restoration and super-resolution algorithms for facilitating improved fusion objectives in the design of efficient target surveillance and tracking mechanisms. An important application area where such algorithms play a critical role is in the design of intelligent integrated processing

architectures for achieving all-weather day-and-night terminal guidance capabilities for smart weapons.

Our investigations towards this goal hence focused on several specific topics that included the following:

- tailoring of iterative statistic-based (Maximum Likelihood, Maximum A Posteriori) super-resolution algorithms by quantifying *a priori* object knowledge;
- evaluation of robustness of super-resolution algorithms for imperfect sensor models and fluctuations in clutter and noise levels;
- design of blind deconvolution algorithms and their implementation for optimized performance;
- evaluation of the role of collecting oversampled data in achieving reliable extension of image bandwidth;
- development of metrics to measure resolution in images and to assess resolution enhancement in processed images.

Due to the fact that investigations on resolution enhancement have both military and non-military applications, the studies were conducted using a general framework and employing mathematical models. However, in order to provide validation of the studies conducted and the results obtained to military environments, the work was performed in close collaboration with the Air Force Wright Laboratory Armament Directorate personnel. In particular, the various simulation experiments conducted to demonstrate the efficiency of the algorithms developed in this project were appropriately tailored to process passive millimeter-wave (PMMW) imagery data acquired from state-of-the-art PMMW radiometers being built by the Advanced Sensors Group at the Air Force Wright Laboratory Armament Directorate.

1.2 Need for Super-resolution of Passive Millimeter-wave Images

Present day military operations that may include execution of tactical surveillance and tracking missions in hostile environments often demand the capability for all-weather day-and-

night sensing and intelligence data collection. Various missions such as reconnaissance, landing and take-off of aircraft, covert deployment of special operations teams, detection and tracking of tactical mobile and extended area high-value targets, and aimpoint selection and precision kill, are typical in their need for the ability to execute the mission in any set of contingencies. Development of sensors operating at different ranges of the electromagnetic spectrum has been the primary technological approach to provide this capability. While two types of sensing, *viz.*, passive infra-red (IR) and active radar, have received a significantly greater attention in the past, sensors operating in the millimeter wavelength regime are being increasingly considered in the recent times.

Passive Millimeter-wave (PMMW) sensing, in particular, combines the strong features of both IR and Synthetic Aperture Radar (SAR) systems. Like IR, it provides a passive mode of sensing and hence is appropriate for covert operations. Since measurable emissions from the scene or target being imaged at wavelengths that are considerably longer than the wavelengths of IR and visible light are utilized, significant all-weather day-and-night imaging capabilities are achieved. These wavelengths are typically in the range of 30 to 120 GHz, corresponding to wavelengths of 10 mm to 2.5 mm. Because these are wavelengths in the microwave band, they will pass through substantial amounts of attenuating media, such as fog, clouds, rain, smoke, dust, clothing, camouflage, and even some building material. Furthermore, being a fundamental emission process, the radiation is available not only on bright sunny days but also in hazy and cloudy environments and even in the darkest of nights. Theory and background on general PMMW radiometry can be readily found in the literature [1,2].

Although PMMW sensing provides the benefits cited above and hence is attractive for all-weather operations, there are several disadvantages the principal one being poor resolution in the acquired imagery. The diffraction-limited angular resolution, θ, of an incoherent imaging system is given by [3]

$$\theta = 1.22\frac{\lambda}{D}$$

where λ is the effective wavelength of imaging and D is the diameter of the limiting aperture of the antenna or lens. As λ increases, the achievable angular resolution decreases, *i.e.* θ, the

size of the angle between two resolvable points, increases. For a typical PMMW sensor with a 1 ft diameter antenna and operating at 95 GHz, the angular resolution is only about 10 mrad, which translates into a spatial resolution of about 10 meters at a distance of 1 Km. Some recent studies have also established that for ensuring reasonably adequate angular resolution (typically of the order of 4 mrad), a 95 GHz PMMW imaging system with a sensor depression angle of 60° - 80° needs to be confined to very low operational altitudes (of the order of 75-100 meters), which puts inordinate demands on the surveillance and guidance schemes to facilitate such requirements. In comparison with optical and IR sensors, since the wavelength of millimeter-wave radiation is 1000 times greater than the wavelength of optical and IR radiation, an aperture of a given diameter D has 1000 times less resolving power than an equivalent optical aperture. Increasing the aperture size by 1000 times to obtain an improved resolution comparable to that of optical and IR sensors is usually infeasible. This in turn demands use of clever image processing techniques to process the acquired imagery data and obtain improved resolution in the processed images.

The fundamental problem underlying the sensing operation is the "low-pass" filtering effect due to the finite size of the antenna or lens that makes up the imaging system and the consequent imposition of the underlying diffraction limits. Hence the image recorded at the output of the imaging system is a low-pass filtered version of the original scene. The portions of the scene that are lost by the imaging system are the fine details (high frequency spectral components) that accurately describe the objects in the scene, which also are critical for reliable detection and classification of targets of interest in the scene. Hence some form of image processing to restore the details and improve the resolution of the image will invariably be needed. Traditional image restoration procedures (based on deconvolution and inverse filtering approaches) attempt mainly at reconstruction of the passband and possibly elimination of effects of additive noise components. These hence have only limited resolution enhancement capabilities. Greater resolution improvements can only be achieved through a class of more sophisticated algorithms, called super-resolution algorithms, which provide not only passband reconstruction but also some degree of **spectral extrapolation**, thus enabling to restore the high frequency spatial amplitude variations relating to the spatial resolution of the sensor and

lost through the filtering effects of the seeker antenna pattern. A tactful utilization of the imaging instrument's characteristics and any *a priori* knowledge of the features of the target together with an appropriately crafted nonlinear processing scheme is what gives the capability to these algorithms for super-resolving the input image by extrapolating beyond the passband range and thus extending the image bandwidth beyond the diffraction limit of the imaging sensor.

For application in surveillance environments, it must be emphasized that super-resolution is a post-processing operation applied to the acquired imagery and consequently is much less expensive compared to improving the imaging system for desired resolution. As an example, it may be noted that for visual imagery acquired from space-borne platforms, some studies indicate that the cost of camera payload increases as the inverse 2.5 power of the resolution. Hence a possible two-fold improvement in resolution by super-resolution processing in this application roughly translates into a reduction in the cost of the sensor by more than 5 times. Similar relations also exist for sensors operating in other spectral ranges (due to the relation between resolution and antenna size), confirming the cost effectiveness of employing super-resolution algorithms. The principal goal of super-resolution processing in a multispectral surveillance environment is hence to obtain an image of a target of interest via post-processing that is equivalent to one acquired through a more expensive larger aperture sensor.

Most of the recent analytical work in the development of image restoration and super-resolution algorithms has been motivated by applications in Radioastronomy and Medical Imaging. While this work has given rise to some mathematically elegant approaches and powerful algorithms, a certain degree of care should be exercised in adapting these approaches and algorithms to tactical surveillance scenarios. This is due to the convergence problems often encountered by iterative schemes and the specific statistical models representing the scenarios facilitating their development. For example, a slowly converging algorithm that ultimately guarantees the best resolution in the processed image may pose no implementational problems in radioastronomy; however, it could be entirely unrealistic for implementation in an autonomous unmanned tactical system that must operate fast enough to track target motion.

Hence a careful tailoring of the processing algorithm is of critical importance in order to realize the possible performance benefits from super-resolution processing which include better false target rejection, improved automatic target recognition and reliable tracking.

1.3 Outline of the Report

The principal objectives in this project are the development of novel super-resolution algorithms for processing PMMW imagery data in order to achieve resolution enhancements that facilitate improved surveillance and tracking of targets of interest. In this report, we shall describe the various studies conducted and the major results obtained towards meeting the project goals.

In Section 2, some background material that is essential for understanding the significance of the tools employed in this study will be briefly reviewed. A mathematical description of the objectives in designing restoration and super-resolution algorithms will be given which provides a model-based framework for the analytical development of restoration schemes. The role of using *a priori* knowledge in the design of algorithms that provide true super-resolution, *i.e.* spectral extrapolation beyond the image passband, will be given a particular focus. Some practical issues that need to be taken into account in the design of satisfactory algorithms will be described. Some metrics that can be used to assess the image quality and the resolution enhancement subsequent to super-resolution processing will also be introduced.

In Section 3 we shall give a detailed derivation of an iterative super-resolution algorithm obtained from employing a Maximum Likelihood (ML) approach and describe its restoration and super-resolution performance. Since computational complexity is of particular concern in the practical implementation of iterative restoration algorithms, we will provide an analysis of processor requirements for implementing the ML algorithm. Results of processing a number of Passive Millimeter-wave (PMMW) images acquired from state-of-the-art radiometers will also be included in this section.

Section 4 describes a number of modified versions of the basic ML algorithm aimed at obtaining optimal performance even under some non-ideal conditions of operation. Since in practical imaging operations, the parameters of the imaging system cannot be assumed to be perfectly known, a modified algorithm that iteratively updates the sensor point spread function in addition to performing an iterative estimation of the restored image pixels will be developed. For use in environments where the signal-to-noise ratio (SNR) is rather poor, some modifications that attempt to suppress noise-induced processing artifacts will be outlined. For suppression of ringing artifacts in the processed image, another modification that incorporates a background-detail separation approach in the estimation process will be described. Yet another modified version which is obtained by maximizing the posterior distribution function and which yields a Maximum A Posteriori (MAP) restoration algorithm will also be described in this section. We shall also present some processing results obtained by employing these modified algorithms in order to illustrate their strong performance features.

The role of oversampled data in the resolution enhancement performance will be given a particular focus in Section 5 and a new progressive upsampling procedure that provides optimized implementations of iterative super-resolution algorithms will be described. An illustration of the performance of this procedure will also be given in this section by processing a few representative images.

Finally in Section 6, we shall present a summary of the results accomplished during this project and offer some concluding remarks. An outline of some useful research directions for extending work on this project will also be given. A complete listing of the programs in C language that implement the algorithms presented here will be given in an appendix at the end of the report.

2. FACTORS THAT INFLUENCE OPTIMAL DESIGN OF SUPER-RESOLUTION ALGORITHMS

2.1 Mathematical Description of Objectives in Image Restoration and Super-resolution

Development of efficient methods for restoration of degraded images has a considerably long history [4] and a number of algorithms derived by employing deterministic and stochastic frameworks, some of which are even suitable for digital implementation, are in existence. In general, these methods attempt to recover the features of an object or a scene that is imaged in an imperfect manner. The objectives in the restoration processing could range from simple noise removal (by appropriate filtering operations) to more elaborate processing that attempts to reverse the blur in the acquired image which may be caused by a number of factors such as relative motion between the object and the imaging sensor and out-of-focus imaging. While these methods generally result in noticeable improvements in the resolution present in the processed image, the primary emphasis in the processing is on passband reconstruction only, *i.e.,* restoring the frequency components within the bandwidth of the sensor. In contrast, super-resolution is a form of image restoration where the primary goal is to recover attenuated frequencies from diffraction limited imaging operations by appropriately extrapolating the image bandwidth beyond the sensor cutoff frequency. Thus the primary objective of super-resolution processing is to obtain a processed image that is equivalent to one obtained from a larger aperture (higher cost) sensor. A more detailed description of the objectives of image restoration and super-resolution from a frequency spectrum reconstruction viewpoint can be found in [5].

For a precise description of the restoration and super-resolution objectives, it is useful to start with a mathematical model for the image formation process. In particular, for obtaining a discrete model of the imaging process to serve as a basis for tailoring a restoration/super-resolution algorithm, it is common to start with the assumption that the blurred image can be represented as the output of a linear shift-invariant system with a finite extent Point Spread Function (PSF) whose input is the scene (or object) that is imaged. Hence,

considering that the image is obtained from an incoherent sensing operation and any noise contamination can be approximated by an additive zero-mean white Gaussian random field which is independent of the object, one can obtain a model for the degraded image in the form

$$g(x,y) = \sum_{(\tilde{x},\tilde{y})\varepsilon R_o} h(x-\tilde{x}, y-\tilde{y}) f(\tilde{x},\tilde{y}) + n(x,y) \tag{1}$$

where $f(\tilde{x},\tilde{y})$ denotes the object's intensity function defined over a region $(\tilde{x},\tilde{y})\varepsilon R_o$, $g(x,y)$ denotes the intensity detected in the image over a region $(x,y)\varepsilon R_I$, $h(z_1,z_2)$ denotes the sensor PSF, and $n(x,y)$ denotes the additive noise contamination. The classical image restoration problem is to find the object intensity estimate $\{\hat{f}(x,y)\}$ given the image data $\{g(x,y)\}$ and knowledge of the PSF function $h(z_1,z_2)$ for the imaging sensor.

Assuming that the image to be processed consists of $M \times M$ equally spaced gray level pixels obtained through a sampling of the image field at a rate that satisfies the Nyquist criterion, and using a lexicographical ordering of the signals g, f and n, one can rewrite Equation (1) as a convolution of two one-dimensional vectors $h = [h(1), h(2), ..., h(N)]^T$ and $f = [f(1), f(2), ..., f(N)]^T$ in the form

$$\begin{aligned} g(i) &= h(i) \otimes f(i) + n(i) \\ &= \sum_{j=1}^{N} h(i-j) f(j) + n(i) \quad , \quad i = 1,2,..,N \end{aligned} \tag{2}$$

where $N = M^2$ and $\otimes$ denotes convolution. More compactly, Equation (2) can be rewritten as the vector equation

$$g = Hf + n \tag{3}$$

where g, f, and n are vectors of dimension N, and H denotes the PSF block matrix whose elements can be constructed from the PSF samples $\{h(1),h(2),...,h(N)\}$. It should be noted that Equations (2) and (3) represent space-domain models and are equivalent to the frequency-domain model

$$G(\omega) = H(\omega)F(\omega) + N(\omega) \tag{4}$$

where ω is the discrete frequency variable, and $G(\omega)$, $F(\omega)$, $H(\omega)$, and $N(\omega)$ are the Discrete Fourier Transforms (DFT's) of the N-point sequences $g(i)$, $f(i)$, $h(i)$, and $n(i)$ respectively.

Diffraction limits underlying practical sensing operations result in acquiring blurred images with inherently poor resolution. The blurring effect caused by the sensor results in the image spectrum $G(\omega)$ being a low pass filtered version of the object spectrum $F(\omega)$, and consequently the higher frequencies present in the object (which are responsible for the spatial resolution) will be attenuated at the cutoff frequency ω_c determined by the diffraction limit of the sensor. The primary goal of super-resolution processing is to extend the image bandwidth beyond ω_c by estimating these attenuated spatial frequencies in the range $\omega_c \le \omega \le \omega_s/2$, where ω_s is the sampling frequency used. An efficient super-resolution algorithm effectively utilizes available knowledge of the imaging sensor's characteristics and any *a priori* knowledge of the scene being imaged to construct the object estimates $\hat{f}^T = \left[\hat{f}(1),\hat{f}(2),...,\hat{f}(N)\right]$ such that

$$\hat{f} = \arg\ \min_f J(g,f) = \arg\ \min_f J\left(g(i) - \sum_j h(i-j)f(j)\right) \tag{5}$$

where $J(g,f)$ is an appropriately chosen norm to measure the closeness of the estimate to the original object. It must be emphasized that the bandwidth extension beyond ω_c serves as a measure of the efficiency of the super-resolution algorithm tailored. One may also note in this

context that traditional image restoration procedures which attempt to implement a convolutional inverse (as for instance, by computing the inverse of the PSF matrix H) are generally successful in correcting the degradations within the passband only, *i.e.* in $0 \leq \omega \leq \omega_c$, while having no significant bandwidth extension capabilities.

2.2 Limits on Super-resolution

The idea of recreating the spectral components that are removed by the imaging process and hence are not present in the image available for processing may pose some conceptual difficulties, which may lead one to suspect whether super-resolution is indeed possible. Fortunately there exist sound mathematical arguments confirming the possibility of spectral extrapolation. The primary justification comes from the Analytic Continuation Theorem and the property that when an object has finite spatial extent its frequency spectrum is analytic [6]. Due to the property that a finite segment of any analytic function in principle determines the whole function uniquely, it can be readily proved that knowledge of the passband spectrum of the object allows a unique continuation of the spectrum beyond the diffraction limit imposed by the imaging system. It must be emphasized that the limited spatial extent of the object is critical in providing this capability for extrapolation in the frequency domain.

The Gerchberg-Papoulis formalism [7,8] identifies a possible iterative procedure for solving the super-resolution problem. This algorithm alternately applies constraints in the space-domain and the frequency-domain in the quest for reconstructing the unknown portion of the frequency spectrum from a limited known portion, *viz.* the passband of the sensor, when *a priori* knowledge of the spatial extent of the object is available (the object values within this extent are not known, however). Each iteration of the algorithm is a two-step procedure that can be summarized as under:

STEP 1-- Transform known passband spectrum to space-domain and apply space limit constraint to set object values to zero outside the region where it is known to be space limited.

STEP 2-- Transform result back to frequency-domain and correct the passband back to the known original values.

These two steps are iteratively applied until a satisfactory estimate of the spectral components being reconstructed is obtained. Analytical support for the convergence of the procedure comes from consideration of the energy of the error function (error between the true spectrum and its estimate). The application of constraints in each iteration, once in the space-domain and the other in the frequency-domain, provides the nonlinear operations to generate the new frequencies needed for super-resolution.

The arguments given above clearly attest to the possibility of spectral extrapolation beyond the sensor cutoff frequency ω_c. The question then is to determine whether there exist any limits on the bandwidth extension possible or whether it is possible to reconstruct the entire spectrum (up to the limit imposed by the sampling rate used, *viz.* $\omega_s / 2$). Evidently, the amount and the quality of bandwidth extension are measures of the performance of a super-resolution algorithm.

The development of limits for signal recovery has been of interest in Communications and Information Sciences for a number of years (Shannon's information transfer limits). In the present context, it is intuitive to expect that the limits on spectrum extrapolation come from a number of factors, most notably those that affect the precision of the data being processed. Clearly, one of the important factors is the noise level present, i.e. SNR of the image. The role of SNR in affecting the quality of image restoration has long been studied. In fact, many of the available approaches for image restoration start from recognizing the problem associated with inverting the imaging equation described earlier in Equation (3), *viz.*,

$$g = Hf + n,$$

as an ill-posed problem, i.e. the object estimate $\hat{f}$ obtained from inverting the above equation is not a continuous function of image data g and arbitrarily small perturbations in g can lead to large variations in the estimate $\hat{f}$, and attempting to devise procedures (such as regularization [9,10]) for overcoming this problem.

Several other factors can equally influence any limits on possible spectrum extrapolation and consequently establishing precise analytical limits to the bandwidth extension possible in all cases seems infeasible. For optical images, by using an information theoretic analysis, Kosarev [11] determined a resolution limit under certain conditions to be

$$r_l = \frac{1}{\omega_c \log_2(1+SNR)} \tag{6}$$

where r_l is the minimum resolvable distance between two point sources in the object, ω_c is the cutoff frequency of the imaging system and *SNR* is the image signal-to-noise ratio. It may be noted that the quantity on the right-hand-side is fixed for any specific image data and hence any separation greater than the value of r_l can be potentially resolved by super-resolution processing. This formula also specifies the role of ω_c in obtaining resolution limits. Note that as ω_c increases, r_l decreases (i.e. resolution increases) which agrees well with intuition.

Another term that affects any limits on spectrum extrapolation is the sampling rate used. Clearly, higher the value of ω_s, larger will be the width of the frequency band $\omega_c \le \omega \le \omega_s/2$ for any imaging system, and greater is the potential for resolution improvement. In fact, many practical super-resolution algorithms use interpolation or other upsampling procedures appropriately during execution of the algorithm to obtain the effect of larger ω_s [12].

One factor that is not readily apparent in regard to its relation to resolution limits is the spatial extent of the object. Very recently, Sementilli et.al. [13] have derived an approximate bound on bandwidth extrapolation for optical images which involves ω_c, ω_s and the object extent. This bound not only establishes that the potential for super-resolution increases with decreasing spatial extent of objects being imaged, but also confirms the intuitive feeling that restoration results often demonstrated for two-point source objects do not necessarily translate into corresponding performances in the case of spatially extended objects.

It should be emphasized that while analytical limits such as the ones discussed above are useful for determining potential benefits from super-resolution processing of given image data, one does not have to be necessarily discouraged if the data to be processed does not meet these limits. Observe that only the characteristics of the image data $\boldsymbol{g}$ and of the imaging system, and only a limited information on the object (*viz.*, spatial extent) are used in developing these limits. If any additional knowledge of the object being restored is available, one may attempt to use this information for possible spectral extrapolation and hence improved resolution. How to utilize this information is at the heart of a well-tailored super-resolution procedure.

2.3 Use of *a priori* Knowledge in Design of Processing Schemes

As noted in the last section, due to the ill-posed nature of the inverse filtering problem underlying image restoration and super-resolution objectives, it is necessary to have some *a priori* information about the ideal solution, i.e. the object f being restored from its image g. In algorithm development, this information is used in defining appropriate constraints on the solution and/or in defining a criterion for the "goodness" of the solution. It may be recalled that the use of such constraints is fundamental in the application of Gerchberg-Papoulis formalism and is in fact the basis for the nonlinear processing underlying super-resolution.

The specific *a priori* knowledge that can be used evidently depends on the specific application. For applications in astronomy, it could come in the form of some known facts about the spectral differences of the objects one is looking for. In medical imaging and in military applications, it could come from the geometrical features of the object (target shape, for instance). Also, one could use the fundamental knowledge that the reflectivity of any point on the ground cannot be negative. In addition to the nonnegativity constraint, a space constraint resulting from the known space-domain limits on the object of interest could be used. Other typically available constraints include level constraints (which impose upper and lower bounds on the intensity estimates $\hat{f}_j$) , smoothness constraints (which force neighboring pixels in the restored image to have similar intensity values) and edge-preserving constraints.

More complicated constraints are possible, but in general they result in tuning the algorithms to specific classes of targets.

Varying by the extent to which *a priori* knowledge can be incorporated in algorithm development, there have been introduced into the literature a large number of image restoration approaches and algorithms too vast to describe or reference here. One may refer to some recent survey papers [14,15] for a review of the extensive activity on this topic. In the context of the objectives of the present project however, it is important to recognize that only a small subset of the approaches that are developed for image restoration have received some interest for their super-resolution capabilities, i.e. possible spectrum extrapolation performance. One may note that not all image restoration methods provide the capability for super-resolving. In fact, a majority of existing schemes may perform decent passband restoration, but provide no bandwidth extension at all.

The various approaches in general attempt to code the *a priori* knowledge to be used by specifying an object model or a set of constraint functions, and further employ an appropriate optimization criterion to guide in the search for the best estimate of the object. A convenient way of classifying the resulting algorithms is into **iterative** and **non-iterative** (or direct) schemes. Figs. 1a and 1b depict schematically these two basic approaches. Non-iterative approaches generally attempt to implement an inverse filtering operation (without necessarily performing the computation of the inverse of the PSF matrix H, however) and have poor noise characteristics. All required computations and any possible use of constraint functions are applied in one step. In contrast, iterative methods apply the constraints in a distributed fashion as the solution progresses and hence the computations at each iteration will be generally less intensive than the single-step computation of non-iterative approaches. Some additional advantages of iterative techniques are that, (1) they are more robust to errors in the modeling of the image formation process (uncertainties in the elements of the PSF matrix H, for instance), (2) the solution process can be better monitored as it progresses, (3) constraints can be utilized to better control the effects of noise (and possibly clutter), and (4) can be tailored to offset sensor nonlinearities. The disadvantages of these methods generally are, (1) increased

computation time if a number of iterations are to be executed for obtaining the required level of resolution enhancement, and (2) need for proving convergence of the iterative scheme. Despite these disadvantages, iterative methods are generally the preferred approach due to their numerous advantages and also since the iteration can be terminated once a solution of a reasonable quality is achieved. Consequently, most of the research in tailoring super-resolution algorithms is taking place at present times in formulating iterative procedures.

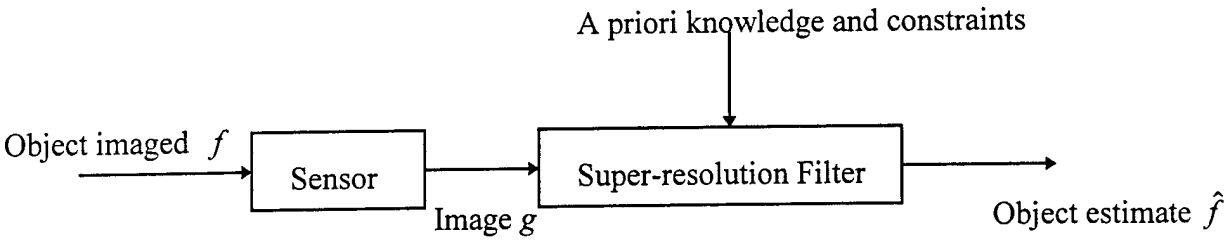


Fig. 1a. Schematic of Noniterative (Direct) Super-resolution

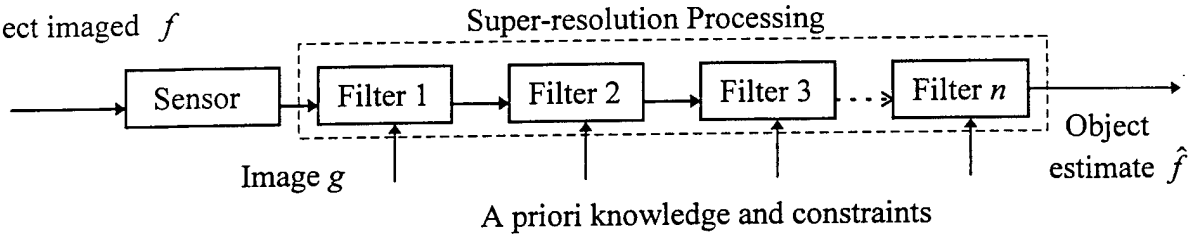


Fig. 1b. Schematic of Iterative Super-resolution

Fig. 1. Iterative and Non-iterative Super-resolution Schemes

2.4 Some Issues of Particular Concern in Design of Super-resolution Algorithms

An important consequence of the ill-posed nature of the deconvolution problem underlying image restoration and super-resolution processing is the generation of artifacts in the processed image. Restoration artifacts are patterns and/or geometrical structures, such as lines, points, etc., that are formed in the restored image which are not present in the object imaged. Several phenomena contribute to the production of artifacts in the restoration process [16] and the intensity of these can magnify as the number of iterations is increased. Presence of artifacts in the processed image is generally undesirable since it can interfere with the overall goals for which restoration and super-resolution processing are included, such as reliable target

detection and recognition, due to the fact that the artifacts could obscure some critical feature of the target, introduce features that may not be present in the target of interest, or otherwise cause distortions of feature maps that may be extracted from the restored image. Consequently, some type of post-filtering and/or modified implementations will generally be needed for eliminating or suppressing the artifacts as they are generated.

An important factor that could impose limits on successful restorations is the noise level present (*i.e.* SNR of the image). As discussed earlier, Kosarev's [11] limit to the achievable spectral extrapolation inversely relates the minimum resolvable distance between two point sources in the scene imaged and the SNR of the obtained image. Although artifact production during image restoration is due to a number of factors, the presence of significant amounts of noise in the measured data (poor SNR levels) is perhaps a major contributing factor [17]. Furthermore, in the super-resolution processing of images that contain high frequency noise, the creation of higher frequencies in the processed image could be a resultant of energy flowing from noise into the image pixels, which evidently needs to be curtailed. Consequently, the dynamics of information transfer between the noise present and the image during successive iterations are of particular importance, and any modifications to the iterative algorithm that take into account these dynamics are of specific interest. This is of particular practical significance in the processing of several sensor-acquired imagery data, such as PMMW images, since typical radiometric images are characterized by SNR levels which are about 20 dB or less.

The artifacts generated due to the presence of noise in the image data manifest themselves in the restored image much differently from the more familiar "ringing artifacts" [18]. The latter are primarily due to Gibbs phenomena resulting from the truncation of Fourier components caused by the extent of the acquired spatial frequencies of the scene or the object imaged being finite. Consequently, during the restoration process, particularly when statistical estimation methods are employed, noticeable distortions near edges occur as sharp transitions, or edges that may be present in the intensity distribution function become more accentuated and emerge with overshoots (or ringing). These artifacts show up in the processed image as

periodic (or decaying periodic) repetitions of sharp intensity transitions and generally become more intense as the number of iterations in super-resolution processing grows.

Commonly used techniques for eliminating restoration artifacts employ some form of filtering of the processed image. While these can reduce the artifacts, they also generally reduce the resolution, counteracting any gains produced from super-resolution processing. Consequently, clever techniques need to be employed for achieving a reduction of the artifacts without the disadvantage of resolution loss. Two such methods that involve a post-filtering adjustment technique and a background-detail separation of the image to process them separately will be described later in this report.

Yet another cause for artifact production that requires attention, particularly in super-resolution processing, is the aliasing of the extrapolated spectral components with the original frequency components present in the passband spectrum of the image data. This is usually the result of an insufficient rate of sampling used for data collection and requires some sophisticated processing steps to eliminate. Since inclusion of these processing steps is fundamental to achieving the goals of super-resolution, this issue will be discussed in a greater detail in a later section.

Although general purpose algorithms for image restoration and super-resolution can be developed without using excessive knowledge of the sensor design and the variations in the sensing process at different wavelengths, some understanding of the sensor phenomenology can be utilized in an attempt to optimize the processing algorithm for the specific imaging modality. It is well known that the contrast behavior of an object that is imaged against its background depends not only on the sensor parameters (such as wavelength of radiation, detector polarization and thermal sensitivity), but also on the object characteristics (such as reflectivity, physical and chemical composition, shape and physical extent) and also on atmospheric conditions. Thus, man-made and natural objects appear with different brightness in the image. For example, in PMMW images, vegetation and bare soil may appear brighter due to their warm physical temperature, whereas metal objects on flat earth reflect the "cold" sky and hence appear in the image with lower brightness.

Some additional considerations are useful in tailoring super-resolution algorithms for PMMW images. In comparison with the contrast available in IR imagery, one may note that black body radiation in the MMW portion of the spectrum is much more dependent on the emissivity/reflectivity of the object of interest than on the temperature of the object. In the MMW range, the emitted energy is a function of T, the object temperature, rather than on T^4 as it is in the IR range of the spectrum [12]. Furthermore, at MMW wavelengths, black body radiation can be described by the Rayleigh-Jeans law (which provides a linear approximation to Planck's law) in the form

$$N_{bb} = \frac{2ckT}{\lambda^4} \tag{7}$$

where c is the speed of light, k is Boltzmann's constant, T is the object's temperature (in Kelvin) and λ is the wavelength. Thus denoting the proportionality constant $\frac{2ck}{\lambda^4} = \tilde{C}$, one can invert this relationship to obtain an expression for the "equivalent black body temperature" of an object for a given radiance level received at the aperture of the sensor as

$$T_{eq} = \frac{N_{bb}}{\tilde{C}}. \tag{8}$$

T_{eq} is usually referred to as "radiometric temperature" in MMW radiometry. Typical atmospheric radiometric temperatures are in the range of $20^0 - 100^0 K$. Although typically more energy is emitted at IR than at MMW by an object, MMW images tend to have a higher contrast than IR images [13]. The higher contrast can be attributed to the fact that cloud cover is comparatively transparent to millimeter waves and the sky temperature can be quite low (typically less than $100^0 K$). Thus an object at $300^0 K$ will have an emissivity that can produce a brighter signature resulting from the cold sky reflections. The effects of image contrast on the PMMW super-resolution performance are not well understood at present. One may note however that as the imaging frequency is increased, the acquired image tends to have a higher spatial resolution for a given antenna size at the cost of reduced image contrast. Thus, quantitative evaluations of processing PMMW images of the same scenes collected at 35 GHz, 95 GHz and 120 GHz could help in understanding the relation between the contrast in acquired imagery and the spectral extrapolation that could be achieved from processing it.

2.5 Image Quality Measures and Assessment of Resolution Improvement

A particularly challenging question in resolution enhancement studies is devising appropriate metrics to assess the quality in a processed image in order to understand the resolution improvement gained from implementing a particular algorithm or to compare the relative performance of two alternate algorithms. This question attains particular significance in the use of iterative restoration and super-resolution algorithms since the resolution gains achieved after a certain number of iterations may not be significant enough to justify the investment in computational effort and hence a criterion to stop the iterations may need to be established based on some quantitative measures or numerical metrics.

Part of the difficulty in assigning a precise metric for assessment of resolution improvements from a given algorithm stems from the uncertainty surrounding the question of how to measure resolution in a given image. Evidently, several definitions are possible depending on what type of information is sought. For example, if separation of point sources in the scene being imaged is of interest, one may attempt to use the standard measure given by the Rayleigh criterion relating the angular resolution θ to the wavelength of radiation λ and the aperture diameter D

$$\theta = 1.22\frac{\lambda}{D} .$$

This expression results from the assumption that two points on the object or scene are resolved in the image produced if their angular separation is such that each point is located at the first minimum of the diffraction pattern of the other [3]. However, if the question of interest is sharpening of the edges and shapes in the image being processed such that specific objects can be more easily recognized, the above measure may not be the right one to use and a different metric that appropriately quantifies the edge sharpening achieved would be needed. Furthermore, for images of scenes consisting of complex extended objects, the Rayleigh measure which is based on the resolution of point sources in the object is clearly not appropriate. In these cases, it is difficult to derive a satisfactory quantitative measure on which to base the decision that objects in the scene are resolved and the decision may need to be based on visual perception which becomes a subjective measure. Also, if the processed image

is used for the extraction of certain features which are then used for some specific purpose (such as target detection, object recognition, or fusion of images), the quality of features that can be extracted becomes the meaningful criterion. Added to the difficulty is also the diversity in the possible measurements since a number of statistical, spectral and spatial metrics can be used to assess a given image.

Since the primary goal of image super-resolution is to restore the image by extending the frequency content beyond the cutoff frequency of the sensor ω_c, it is natural that any quality metric used to score a super-resolution algorithm must include the size of the extension achieved. However, such a measurement is possible when the ideal image of the object being imaged (that contains the entire spectrum) is available for comparison. In the more practical cases where the ideal image with which the processed image can be compared is not available, one may need to use some indirect ways of quantifying the improvement in the size of the spectrum achieved from super-resolution processing. Consequently, in the further discussion we will distinguish between these two cases.

When the ideal image is available for comparison, various distance metrics can be readily postulated to compare the images and their spectra. Straightforward measures are the L_p-norms of the deviations in the image-domain or in the frequency-domain given by

$$L_p\left(f,\hat{f}\right)=\left(\frac{1}{MN}\sum_{x',y'}\left|f-\hat{f}\right|^p\right)^{\frac{1}{p}} \tag{9}$$

or

$$L_p\left(F,\hat{F}\right)=\left(\frac{1}{MN}\sum_{x',y'}\left|F-\hat{F}\right|^p\right)^{\frac{1}{p}} \tag{10}$$

where f denotes the ideal image of size M by N, $\hat{f}$ the super-resolved image of the same size, and F and $\hat{F}$ denote the corresponding spectra. A problem of particular concern when using distance measures of the above types is that elements with very high or very low magnitudes can unduly influence the value of the metric computed. Consequently, a measure of the correlation between the two images being compared may be more appropriate. Especially

useful is a measure of the similarity between the images and/or their spectra obtained from comparing blocks of pixels from which a "Similarity Map" may be constructed as follows:

1. Each of the two images to be compared, of size M by N pixels, is subdivided into a set of smaller images, each of size K by L ($\alpha\, K = M,\ \alpha \in Integer$ and $\beta\, L = N,\ \beta \in Integer$). The smaller image blocks do not overlap but completely cover the original image
2. Then, KL similarity values are computed by evaluating the cross-correlation between the corresponding image blocks in the ideal image and in the processed image which is defined by

$$C_P^m = \left| \frac{\frac{1}{KL}\sum_{i=1}^{KL}\left(f_R^m(i)-\bar{f}_R^m\right)\left(f_P^m(i)-\bar{f}_P^m\right)^*}{\sqrt{\frac{1}{KL}\sum_{i=1}^{KL}\left(f_P^m(i)-\bar{f}_P^m\right)\left(f_P^m(i)-\bar{f}_P^m\right)^*}\sqrt{\frac{1}{KL}\sum_{i=1}^{KL}\left(f_R^m(i)-\bar{f}_R^m\right)\left(f_R^m(i)-\bar{f}_R^m\right)^*}} \right| \qquad (11)$$

   where $f_R^m(i)$ is the i^{th} component of the point that represents the m^{th} block obtained from the reference image, and $f_P^m(i)$ is the corresponding block obtained from the processed image.
3. The resulting KL similarity values are displayed using the same metric that orders the set of smaller images. In order to obtain a similarity map with as many dimensions as the two images being compared, an M by N matrix can be divided in the same manner as the reference and the processed images are divided. Then, the similarity value obtained from each pair of smaller images can be copied into all the pixels of the corresponding smaller matrix of the similarity map.

The similarity map obtained as above can then be used to compute various metrics that can yield different comparison scores. One particular example of using such a Similarity Map obtained from comparing the spectra of the ideal reference image and the processed image in order to compute a "virtual aperture diameter" that can effectively compare two alternate super-resolution algorithms will be described in a later section.

When the ideal image is not available for comparison, development of appropriate metrics becomes more difficult and should necessarily be based on using the original image being processed in the comparison process. In comparing statistic-based restoration procedures, such as the ML and MAP algorithms, certain statistical parameters can yield a mechanism for useful comparison. Some examples are the likelihood measure given by

$$L\left(g,h,\hat{f}\right)=p\left(g/h,\hat{f}\right)=\prod_{x',y'}\frac{\left(h\otimes\hat{f}\right)^{g}e^{-h\otimes\hat{f}}}{g!} \tag{12}$$

and the posterior distribution given by

$$L\left(g,h,\hat{f},\bar{f}\right)=p\left(g/h,\hat{f},\bar{f}\right)=\left(\prod_{x',y'}\frac{\left(h\otimes\hat{f}\right)^{g}e^{-h\otimes\hat{f}}}{g!}\right)\left(\prod_{x',y'}\frac{\left(\bar{f}\right)^{\hat{f}}e^{-\bar{f}}}{\hat{f}!}\right). \tag{13}$$

In the above expressions, g denotes the acquired image being processed, h denotes the sensor PSF, $\hat{f}$ is the processed image, and $\bar{f}$ is the image prior. Alternately, one can use the value of the "residual" given by

$$R\left(g,h,\hat{f}\right)=\left(\frac{1}{MN}\sum_{x',y'}\left|g-h\otimes\hat{f}\right|^{2}\right)^{\frac{1}{2}}. \tag{14}$$

Yet another way of comparing the restored image with the original image is to extract a slice of data from their spectra and comparing the intensity levels at specified frequency values. More details on such a comparison will be given in the next section when evaluating the performance of the ML super-resolution algorithm. It is to be noted however that such evaluations are only of a subjective nature since an arbitrary determination of which frequency values are to be used for comparison will have to be made.

3. RESTORATION AND SUPER-RESOLUTION PERFORMANCE OF ITERATIVE ALGORITHMS OBTAINED BY A MAXIMUM LIKELIHOOD APPROACH

3.1 Derivation of an Updating Rule for Iterative Maximization of Likelihood

An important class of algorithms that are receiving particular attention in recent times for their super-resolution capabilities are those that can be developed starting with a statistical modeling of the imaging process. The basic idea underlying these methods is to account for the statistical behavior of emitted radiation at the level of individual photon events by constructing appropriate object radiance distribution models (using knowledge of fluctuation statistics). A particularly attractive approach is to obtain a maximum likelihood (ML) estimate $\{\hat{f}(x,y)\}$ *i.e.* the object intensity estimate that most likely have created the measured data $\{g(x,y)\}$ with the PSF process $\{h(z_1,z_2)\}$, which in turn is developed by maximizing an appropriately modeled likelihood function (or the logarithm of this function, for simplicity), *i.e.* obtaining $\hat{f}$ that solves the maximization problem

$$\hat{f} = \arg\max_{f} p(g/f). \tag{15}$$

Modeling the likelihood function is basically obtaining a goodness-of-fit (GOF) quantity for the measured data, since the likelihood function is a statistical distribution function $p(g/f)$ obtained as a fit to the relation between the data $\{g(x,y)\}$ and the object $\{f(\tilde{x},\tilde{y})\}$. The success of image restoration in a given application depends on how good the assumed conditional probability function fits the input/output characteristics of the imaging system.

Selection of an appropriate distribution function for modeling the likelihood $p(g/f)$ should also be based on the simplicity of the resulting algorithm for practical implementations. A particularly simple iteration rule can be developed by modeling $p(g/f)$ as a Poisson

distribution. Furthermore, by assuming that the individual pixels are statistically independent, one can obtain a model for $p(g/f)$ as

$$p(g/f) = \prod_{x',y'} \frac{(h \otimes f)^g e^{-h \otimes f}}{g!}$$
$$= \prod_{x',y'} \frac{\left(\sum_{x'',y''} h(x'-x'',y'-y'')f(x'',y'') \right)^{g(x',y')} \; e^{-\sum_{x'',y''} h(x'-x'',y'-y'')f(x'',y'')}}{g(x',y')!} \tag{16}$$

For maximizing the likelihood one may differentiate $p(g/f)$ with respect to f and solve the equation $\partial p / \partial f = 0$. However, considerable simplification is obtained by attempting to maximize the natural logarithm of the likelihood function $L(g/f) = \ln p(g/f)$. Hence, computing

$$\frac{\partial}{\partial f} L(g/f)\Big|_{x,y} = \frac{\partial}{\partial f} \left\{ \ln \left(\prod_{x',y'} \frac{(h \otimes f)^g e^{-h \otimes f}}{g!} \right) \right\}$$
$$= \frac{\partial}{\partial f(x,y)} \left\{ \ln \left(\prod_{x',y'} \frac{\left(\sum_{x'',y''} h(x'-x'',y'-y'')f(x'',y'') \right)^{g(x',y')} \; e^{-\sum_{x'',y''} h(x'-x'',y'-y'')f(x'',y'')}}{g(x',y')!} \right) \right\}$$
$$= \frac{\partial}{\partial f(x,y)} \sum_{x',y'} g(x',y') \ln \left(\sum_{x'',y''} h(x'-x'',y'-y'')f(x'',y'') \right)$$
$$- \frac{\partial}{\partial f(x,y)} \sum_{x',y'} \sum_{x'',y''} h(x'-x'',y'-y'')f(x'',y'')$$
$$- \frac{\partial}{\partial f(x,y)} \sum_{x',y'} \ln g(x',y')!$$
$$= \sum_{x',y'} \frac{g(x',y')}{\sum_{x'',y''} h(x'-x'',y'-y'')f(x'',y'')} h(x'-x,y'-y) - \sum_{x,y} h(x'-x,y'-y) \tag{17}$$

and setting it to zero, one obtains the extremum condition

$$\sum_{x',y'} \frac{g(x',y')}{\sum_{x'',y''} h(x'-x'',y'-y'')f(x'',y'')} h(x'-x,y'-y) - \sum_{x',y'} h(x'-x,y'-y) = 0 \tag{18}$$

Equation (18) needs to be solved for f with the given g and h. Due to the complexity of the equation, one attempts to obtain an iterative solution in the form given by

$$\hat{f}^{n+1}(x,y) = F\left(\hat{f}^{n}(x,y)\right) \tag{19}$$

where $F(\bullet)$ is the updating function.

Multiplying both sides of Equation (18) by $f(x,y)$ and rearranging terms one obtains

$$f(x,y)\sum_{x',y'} h(x'-x,y'-y) = f(x,y)\sum_{x',y'} \frac{g(x',y')}{\sum_{x'',y''} h(x'-x'',y'-y'')f(x'',y'')} h(x'-x,y'-y)$$

$$f(x,y) = \frac{1}{\sum_{x',y'} h(x'-x,y'-y)} f(x,y)\sum_{x',y'} \frac{g(x',y')}{\sum_{x'',y''} h(x'-x'',y'-y'')f(x'',y'')} h(x'-x,y'-y) \tag{20}$$

Now, using Picard's iteration it is possible to write the expression for the updating rule in the form

$$\hat{f}^{n+1}(x,y) = \alpha\, \hat{f}^{n}(x,y)\left(\frac{g(x,y)}{(h \otimes \hat{f}^{n})(x,y)} * h(x,y)\right) \tag{21}$$

where

$$\alpha = \frac{1}{\sum_{x',y'} h(x'-x,y'-y)} \tag{22}$$

and the symbol $*$ denotes correlation.

If the PSF of the imaging system is symmetric and is normalized such that the sum of its values is one, *i.e.*

$$\sum_{x',y'} h(x'-x, y'-y) = 1$$
$$h(x'-x, y'-y) = h(x-x', y-y') \tag{23}$$

the above iteration can be rewritten in the final form

$$\hat{f}^{n+1}(x,y) = \hat{f}^{n}(x,y)\left(\frac{g(x,y)}{(h \otimes \hat{f}^{n})(x,y)} \otimes h(x,y)\right) \tag{24}$$

which yields a systematic procedure for constructing the object estimate $\hat{f}^{n}(x,y)$ from the given image $g(x,y)$ and the PSF function $h(x,y)$. For commencing the iteration, one can use the starting estimate $\hat{f}^{0}(x,y) = g(x,y)$.

3.2 Evaluation of Restoration and Super-resolution Performance

In this section we shall present the results from a few simple experiments of restoring blurred objects using the iterative rule described by Equation (24). Since our interest in this project is not merely in the restoration within passband but also extrapolation of the frequencies the changes to the image spectrum due to the application of the algorithm will be given a special emphasis.

Experiment 1:

Fig. 2a shows a one-dimensional object characterized by several edges. Fig. 2b shows the image formed when blurred with a sensor with a cutoff frequency 21. Fig. 2c shows the reconstructed image after 90 iterations of the ML algorithm described by Equation (24). While the edge reconstruction and the restoration of the original object are clearly seen, the spectrum extrapolation (and hence super-resolution) can be more clearly seen by examining the

frequency spectra of the signals before and after processing. Fig. 2d shows the spectrum of the original object, Fig. 2e shows the spectrum of the image formed (note the cutoff frequency 21 of the sensor) and Fig. 2f shows the spectrum of the reconstructed image after 90 iterations of the ML algorithm. It should be noted that in these figures only the portion of the spectrum in the high frequency range is plotted to an expanded scale since the signal has low frequency components with relatively large magnitudes that prevent the high frequency portion of the spectrum from being displayed effectively on the same graph. The extrapolation of frequency components beyond the cutoff of 21 clearly demonstrates that this algorithm is super-resolving.

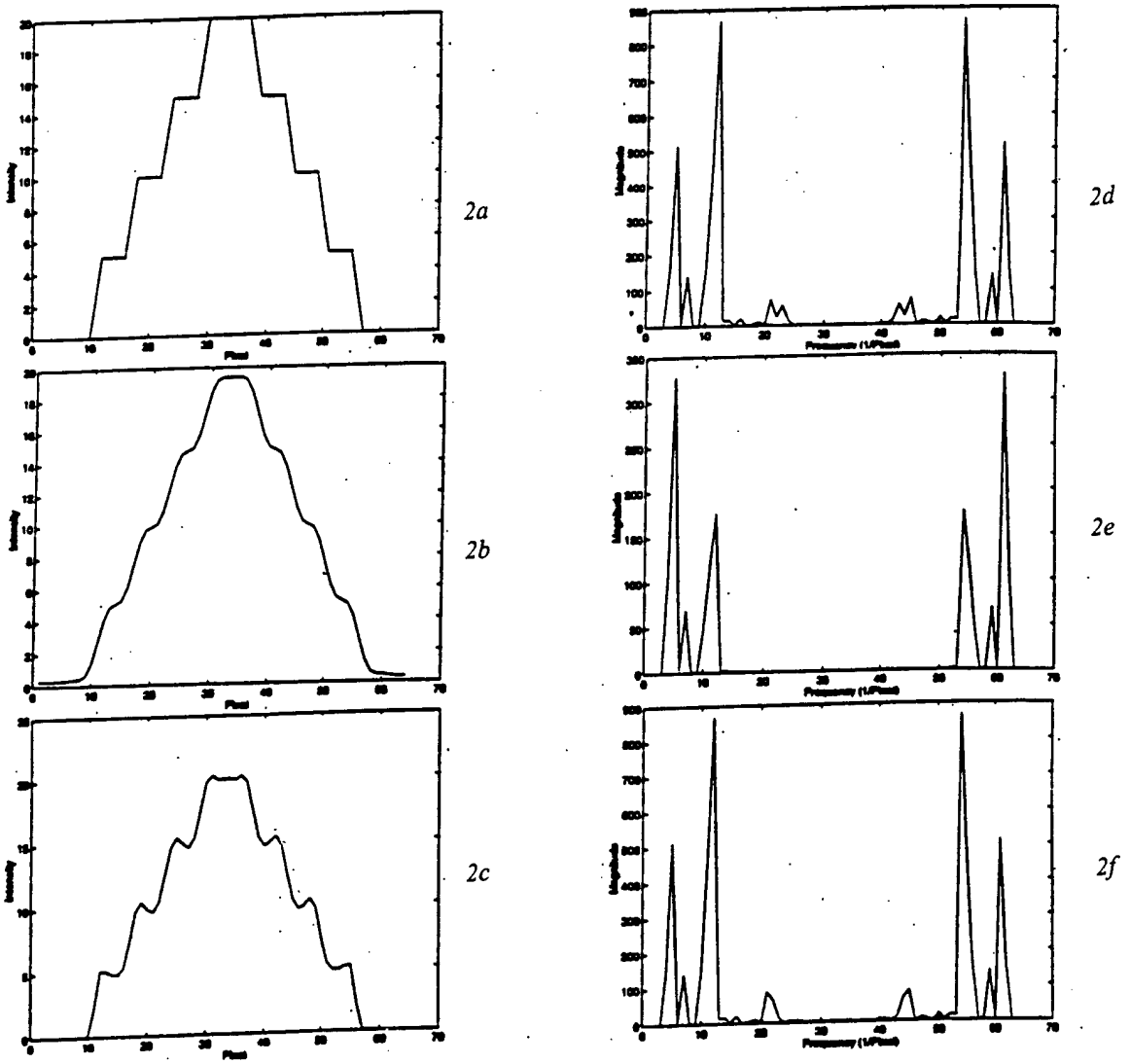
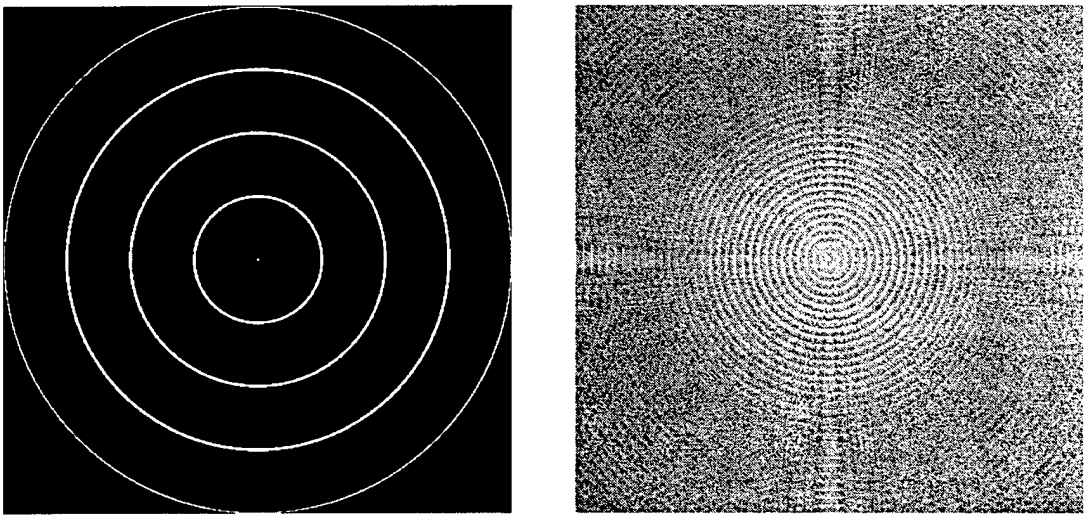


Fig. 2a-2f. Results of processing one-dimensional signal in Experiment 1.

Experiment 2:

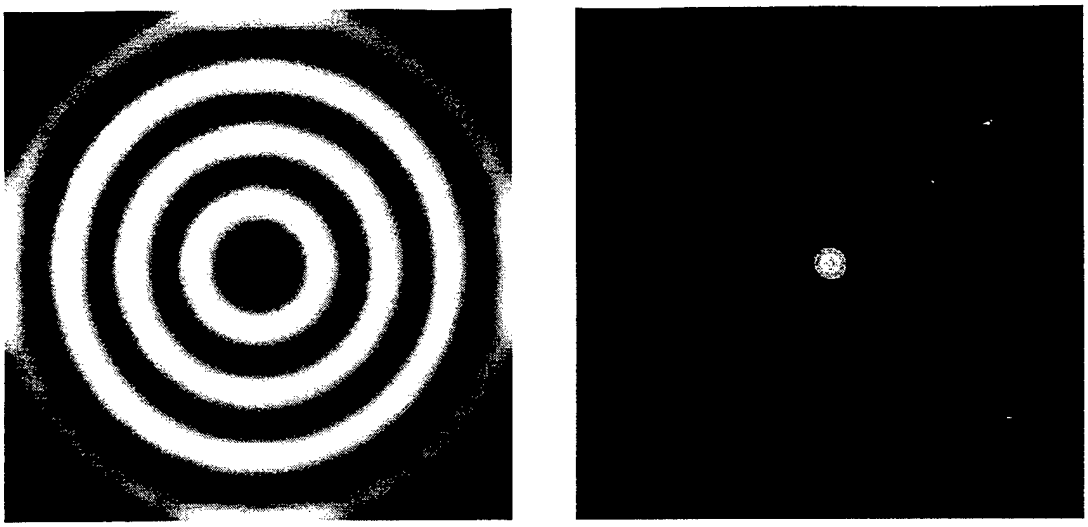
For illustrating the convergence behavior of the iterations and the performance of the ML algorithm given by Equation (24) in processing a two-dimensional image, a controlled experiment was conducted on a simulated object comprising of a series of concentric disks with the background (dark) at intensity value zero and the disks (bright) at intensity value one, shown in Fig. 3a. For simulating the blurring caused by a diffraction limited imaging sensor, this object was convolved with the PSF of a low-pass filter that simulates a sensor with a circular aperture of diameter 16 pixels. The blurred image is shown in Fig. 4a. The effects of this blurring operation can be seen by comparing the spectra shown in Figs. 3b and 4b, which display the extent of frequencies contained in the original object of Fig. 3a and the blurred image in Fig. 4a respectively. To facilitate a digital restoration processing, the blurred image in Fig. 4a is represented on a 512×512 grid of samples, which evidently constitutes an oversampling of this image (since no frequency components beyond the cutoff frequency of 16 are present).



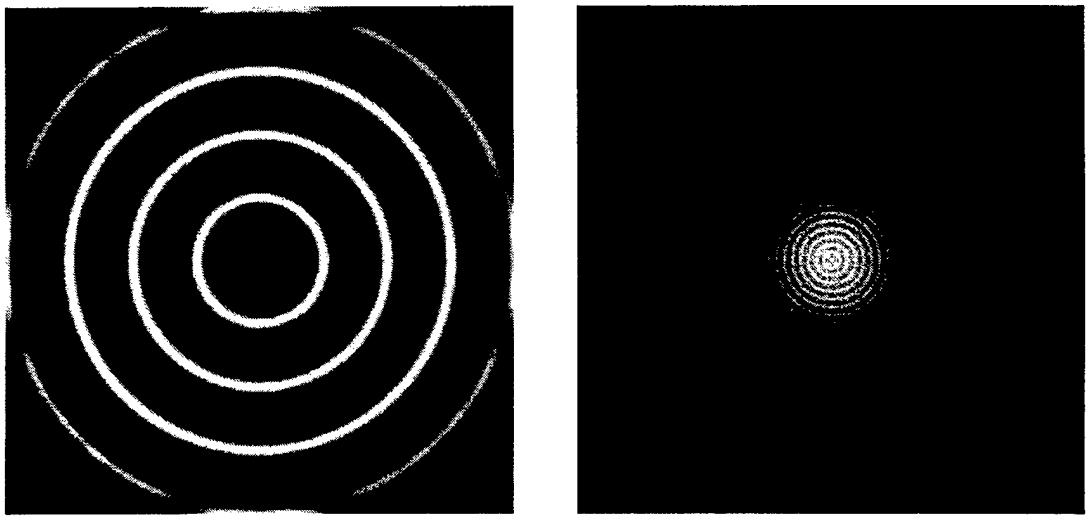
Figs. 3a and 3b. Object represented on a 512 × 512 grid and its spectrum

Fig. 5a shows the restored image obtained after 50 iterations of processing the blurred image with the ML algorithm given by Equation (24). The removal of the blurring effects are clearly evident. However, of particular interest also is the super-resolution performance of the

algorithm, viz. the extrapolation of the image spectrum. The spectrum of this restored image is shown in Fig. 5b which clearly depicts the enlargement of the region that contains the spectral



Figs. 4a and 4b. Blurred image and its spectrum



Figs. 5a and 5b. ML restored image and its spectrum

components, thus verifying the super-resolution capabilities of the algorithm. For obtaining an understanding of the convergence behavior of the algorithm, the processing was continued for 1000 iterations and the normalized restoration error given by the L_2-norm of the deviation $f(x,y) - \hat{f}(x,y)$ was examined. Fig. 6 depicts the behavior of the restoration error as the number of iterations is increased.

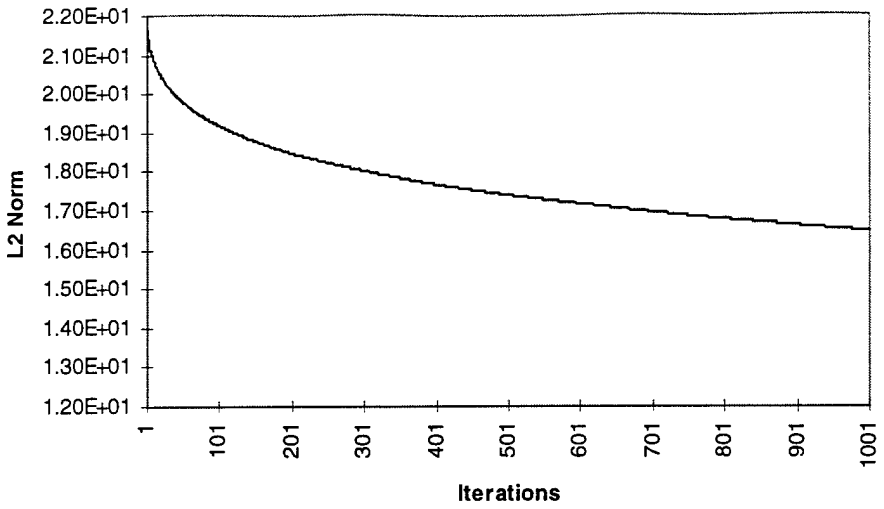


Fig. 6. Convergence of restoration error

A quantitative evaluation of the restoration gain can also be made in this case by computing the "virtual aperture" that likely will acquire the image that resulted from the signal processing. To enable this computation one may obtain the Cross-correlation Similarity Maps comparing the spectra of the unprocessed and processed images with the spectrum of the original ideal object as outlined in Section 2.5. Figs. 7a and 7b respectively show these similarity maps in the Fourier-domain obtained by comparing the blurred image and the ML super-resolved image after 100 iterations with the ideal object. In these images, a value of zero is shown with black color and the images are scaled such that the maximum value is assigned as pure white. One particular advantage of these maps is that they allow to precisely point out those parts of the spectrum that have been restored (white spots) and also the parts where the processing has deteriorated the spectrum (black spots).

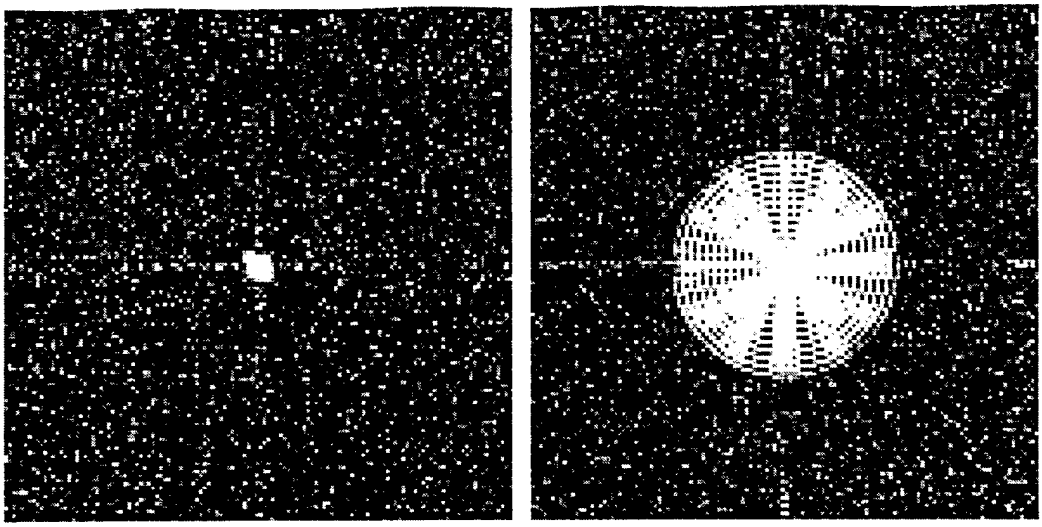


Fig. 7a and 7b. Cross-correlation similarity maps in Fourier-domain

For an imaging system with a circular aperture, the frequency components of any acquired image should reflect this fact by having all information within a circle of radius equal to the cutoff frequency. For the blurred image shown in Fig. 4, this value is straightforward: the cutoff frequency is twice the radius of the aperture function used to blur the original object shown in Fig. 3. For the super-resolved image one can define a corresponding parameter as the cut-off frequency imposed by a "virtual camera" that is needed to acquire an image equivalent to this one, which can in turn be quantified by computing the Virtual Aperture Diameter (VAD). One way of measuring the VAD will be described in the following. In order to evaluate how the extent of similarity is varying along the frequency-axis, an annular ring with width of a certain chosen value (say 11 pixels) is centered in the similarity map. Pixels lying within this ring are weighted by their energies (square of the intensity values) and then averaged. Now, as the radius of the ring is progressively increased from the center of the similarity map, the averages computed will yield a plot that describes how the average similarity is varying as a function of the frequency. Such plots are shown in Fig. 8 where the bottom curve shows the variation of the average similarity as a function of frequency for the blurred image while the top curve shows the corresponding variation for the super-resolved image.

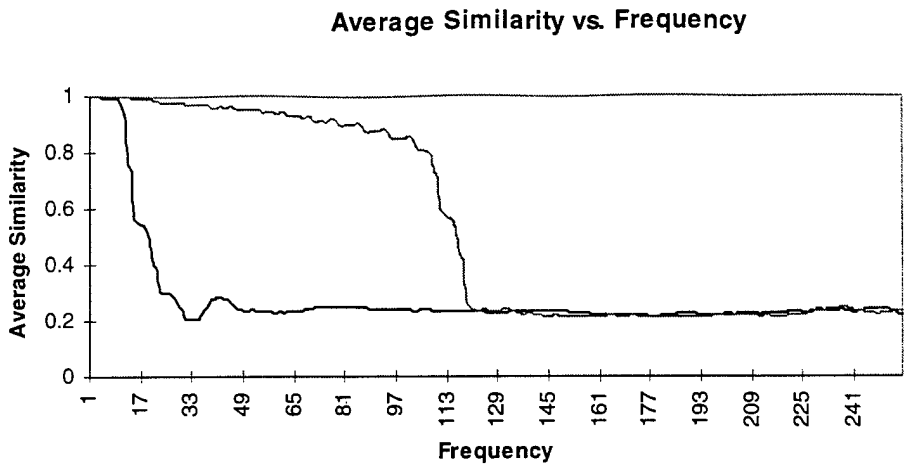


Fig 8. variation of average similarity vs. frequency

The extended cutoff frequency and the VAD can now be readily calculated. Starting with the cutoff frequency of 16 for the blurring employed, one observes that the average similarity value at this frequency is 0.5475. For the super-resolved image, the highest frequency at which an average similarity value no less than 0.5475 can be determined as 115, which gives the value of VAD in this case. It may be noted that an improvement by a factor of 7.1875 is achieved with respect to the original sensor aperture size of 16 pixels due to the ML processing.

Experiment 3:

To evaluate the super-resolution performance of the algorithm on a real degraded image an experiment was conducted by processing a 256×256 image from our database. Fig. 9a shows the original image ("Lena") used in this experiment. Fig. 9b shows the blurred image obtained by convolution with the PSF of a sensor with cutoff $\omega_c = 63$ (which is approximately one-half of the folding frequency and is typical of practical imaging operations). The restoration performance of ML algorithm as the number of iterations is gradually increased is shown in the next set of figures (Figs. 9c and 9d where the restored images after 10 and 100 iterations are shown). The resolution enhancement with only 10 iterations of the algorithm is clearly visible. The algorithm was stopped at the end of 100 iterations since the resolution is comparable to that of the original (unblurred) image. Figures 9e - 9h show the power spectra of various signals; the frequency extrapolation performed by the ML algorithm to achieve

super-resolution is clearly noticeable by comparing the four corners in Figs. 9g and 9h which show the power spectra of the blurred image and the reconstructed image after 100 iterations of the algorithm.

Fig. 9a. Original object

Fig. 9b. Blurred image

Fig. 9c. Processed image after 10 ML iterations

Fig. 9d. Processed image after 100 ML iterations

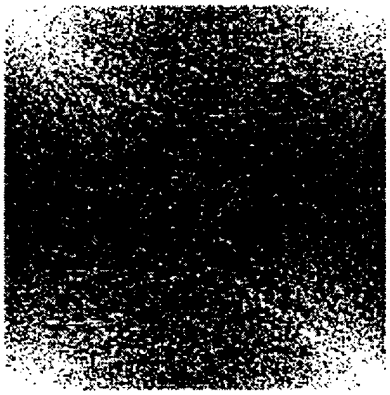


Fig. 9e. Spectrum of object

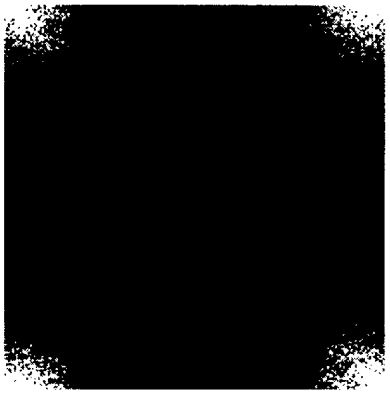


Fig. 9f. Spectrum of blurred image

Fig. 9. Results of processing two-dimensional image in Experiment 3.

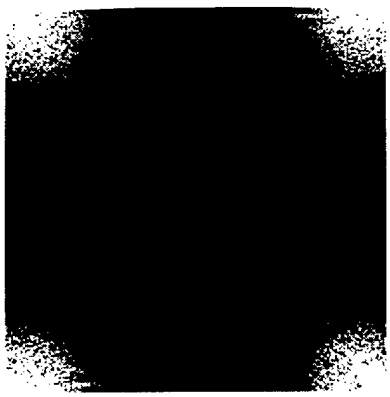


Fig. 9g. Spectrum of processed image after 10 Iterations

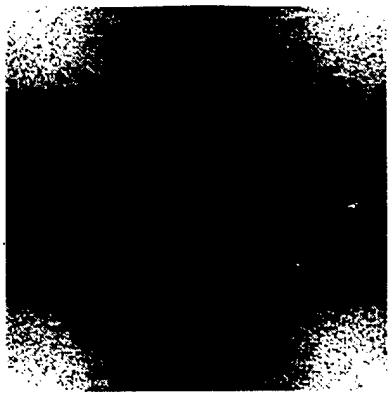


Fig. 9h. Spectrum of processed image after 100 iterations

Fig. 9 (continued). Results of processing two-dimensional image in Experiment 3.

3.3 Analytical Properties of the ML Iteration Algorithm

When iterative restoration algorithms are to be considered for practical implementations, a very important issue is the convergence of iterations. Unlike in the case of other iterative algorithms for image restoration and super-resolution, some convergence results for the ML algorithm given by Equation (24) can be analytically proven. For instance, computing the second derivative of the log-likelihood function $L(g/f)$,

$$\frac{\partial^2}{\partial f^2} L(g/f)\bigg|_{x,y} = \frac{\partial}{\partial f}\left\{\frac{\partial}{\partial f}\ln p(g/f)\right\}\bigg|_{x,y}$$

$$= \frac{\partial}{\partial f(x,y)}\left\{\sum_{x',y'}\frac{g(x',y')}{\sum_{x'',y''}h(x'-x'',y'-y'')f(x'',y'')}h(x'-x,y'-y) - \sum_{x,y}h(x'-x,y'-y)\right\}$$

$$= \sum_{x',y'} g(x',y')h(x'-x,y'-y)\frac{\partial}{\partial f(x,y)}\left\{\frac{1}{\sum_{x'',y''}h(x'-x'',y'-y'')f(x'',y'')}\right\}$$

$$= \sum_{x',y'} g(x',y')h(x'-x,y'-y)\frac{-h(x'-x,y'-y)}{\left(\sum_{x'',y''}h(x'-x'',y'-y'')f(x'',y'')\right)^2}$$

$$= -\sum_{x',y'} \frac{g(x',y')h(x'-x,y'-y)^2}{\left(\sum_{x'',y''} h(x'-x'',y'-y'')f(x'',y'')\right)^2} \tag{25}$$

The quantity on the right hand side is clearly negative, since the image intensity values $g(x',y')$ are always non-negative. This result implies that $L(g/f)$ and $p(g/f)$ are monotonically increasing functions with respect to $f(x,y)$, *i.e.* the likelihood improves with each successive iteration.

Although convergence of the ML iterations can be proven as above, the rate of convergence depends on a number of factors, most notably on the accuracy with which the PSF of the imaging process $h(x,y)$ is known and the Signal-to-Noise Ratio (SNR) of the image being processed. It is usually observed in practice that after a steep convergence during the first few iterations, the later iterations tend to get much slower in the sense of diminishing restoration quality improvement following each iteration. Furthermore, if significant amounts of noise are present or if significant inaccuracies in the assumed PSF exist, the restoration artifacts grow progressively with each iteration resulting in poor convergence (or even divergence) in some cases. Consequently, one will invariably require the termination of the algorithm after a certain number of iteration steps or the implementation of appropriate modifications such as PSF correction (leading to Blind ML Deconvolution) and post-filtering steps as will be described in a later section.

The ML iterative algorithm described by Equation (24) was previously studied for its passband restoration properties and attracted considerable notice by the radio-astronomy and medical imaging communities [21-23]. Very recently, however, the super-resolution capabilities of this algorithm (*i.e.* bandwidth extrapolation properties) are being increasingly studied, specifically for processing diffraction limited imagery acquired in military surveillance and guidance applications [24-26]. The super-resolution capabilities stem from the nonlinear processing involved in each iteration step of the algorithm. Furthermore, the required updating

inherently implements the non-negativity of the object estimate, which is an *a priori* information about the object that needs to be imposed as a constraint in other iterative restoration algorithms.

The ML iterations specified by Equation (24) result in a nonlinear dynamical scheme of the form

$$\hat{f}_{k+1}(j) = \Phi\left(\hat{f}_k(j)\right) \tag{26}$$

where $\Phi\left(\hat{f}_k(j)\right) = \hat{f}_k(j)\left[\left\{\frac{g(j)}{\hat{f}_k(j) \otimes h(j)}\right\} \otimes h(j)\right]$. The convergence of the iterations can hence be examined in terms of the equilibrium points of this dynamical system, or equivalently of the "fixed points" $\hat{f}_k(j)$ that satisfy

$$\Phi\left(\hat{f}_k(j)\right) = \hat{f}_k(j). \tag{27}$$

It is easy to see that this condition is attained at the k-th iteration when the restored image satisfies the equality

$$g(j) = \hat{f}_k(j) \otimes h(j),$$

since the quantity inside the square brackets in the description of $\Phi\left(\hat{f}_k(j)\right)$ becomes 1 (when the PSF is symmetric and normalized, which in turn implies that $1 \otimes h(j) = 1$). Let the image estimate vector $\left\{\hat{f}^*(j)\right\}$ denote this converged value. Thus

$$g(j) = \hat{f}^*(j) \otimes h(j) \tag{28}$$

specifies the fixed points $\left\{\hat{f}^*(j)\right\}$ of the ML restoration algorithm given by Equation (24).

A careful examination of the condition specified by Equation (28) reveals several interesting facts. First of all observe that Equation (28) corresponds to the condition in the frequency domain

$$G(j\omega) = \hat{F}^*(j\omega)H(j\omega) \tag{29}$$

and hence is satisfied by any estimate $\{\hat{f}^*(j)\}$ that restores the passband spectrum. Due to the band-limiting operation of the convolution with the PSF vector $\{h(j)\}$, the image estimate $\{\hat{f}^*(j)\}$ that satisfies Equation (28) is not unique. Since equality outside the passband of the sensor is trivially satisfied (*i.e.,* both the left and the right sides of Equation (29) become zero), there exist an infinite number of vectors $\{\hat{f}^*(j)\}$ that make Equation (28) hold. These vectors are however related in the sense that each of them has a specific structure that permits it to be written as the sum of a fixed vector $\{\hat{f}^*_{pb}(j)\}$ that matches the object spectrum $F(j\omega)$ identically below the diffraction limit (*i.e.,* for $\omega \le \omega_c$) and another vector $\{\hat{f}^*_{sb}(j)\}$ with arbitrary values but with the spectrum below the diffraction limit identically zero. In other words, any estimate $\{\hat{f}^*(j)\}$ that can be written as

$$\hat{f}^*(j) = \hat{f}^*_{pb}(j) + \hat{f}^*_{sb}(j) \tag{30}$$

where $\hat{f}^*_{pb}(j)$ and $\hat{f}^*_{sb}(j)$ satisfy the frequency-domain conditions:

$$\begin{aligned} &\text{(i)} \quad H(j\omega)\hat{F}^*_{pb}(j\omega) = G(j\omega) \\ &\text{(ii)} \quad \hat{F}^*_{sb}(j\omega) = 0 \qquad \text{for } \omega \le \omega_c, \end{aligned} \tag{31}$$

serves as a fixed point of the ML algorithm.

Bayesian estimation methods in general, and ML algorithm in particular, that iteratively attempt to maximize the likelihood of the estimate $\{\hat{f}_k(j)\}$, which gives rise to the image vector $\{g(j)\}$ upon convolution with the PSF vector $\{h(j)\}$, attempt to enforce the passband restoration condition given by (i) above. Hence, when an estimate $\{\hat{f}_k(j)\}$ that satisfies this condition is obtained at the k-th iteration, the algorithm ceases to do any further work. Consequently, convergence of the algorithm is to be interpreted only in terms of restoration of the passband spectrum of the image. In other words, these algorithms principally strive to restore the passband as accurately as possible with the following resultant effects:

(i) Since there is no special incentive for obtaining the part $\{\hat{f}_k^{sb}(j)\}$ that matches the object spectrum outside the diffraction limit specified by ω_c, this vector obtained at the end of several iterations (when convergence has occurred) could have elements whose contribution to the frequency spectrum is much smaller than the contribution of the other part $\{\hat{f}_{pb}^{*}(j)\}$. In other words, the "energy in the frequency extension" could be very small (or even negligible).

(ii) When passband restoration is completed (*i.e.*, when convergence has occurred), no further changes in the estimate $\{\hat{f}_k(j)\}$ are likely, since the likelihood function will have attained its maximum value.

(iii) Any tendencies of the later iterations of the algorithm to degrade the passband restoration performance are resisted by the algorithm, since the algorithm inherently is a convergent one that converges to one of the several fixed points that provide a perfect match of the passband spectrum.

The analysis given above on the convergence behavior of the ML iterations is very helpful in obtaining an insight into the nature of the multiple fixed points of the algorithm. Due to the infinite number of fixed points and the corresponding trajectories that describe the convergence of the initial estimates to these equilibrium conditions, obtaining any quantitative estimates of the convergence rates is rather difficult. The problem can be somewhat simplified

by examining the dynamics of the "iteration error', or change in the estimate during successive iteration steps, defined by

$$e_k(j) = \hat{f}_k(j) - \hat{f}_{k-1}(j) . \tag{32}$$

The behavior of this error is governed by the nonlinear dynamical process

$$e_{k+1}(j) = \hat{f}_{k+1}(j) - \hat{f}_k(j) = \Phi\left(\hat{f}_k(j)\right) - \hat{f}_k(j) \tag{33}$$

which can be re-written as

$$e_{k+1}(j) = \Psi\left(e_k(j)\right) \tag{34}$$

by expressing $\hat{f}_k(j)$ in terms of $e_k(j)$. Equation (34) represents a nonlinear dynamical system whose trajectory behavior starting from an initial state $e_0(j) = \hat{f}_0(j) = g(j)$ is of interest. The equilibrium points of this "error system" are at the locations where $\Psi\left(e_k(j)\right) = 0$, which implies $\Phi\left(\hat{f}_k(j)\right) - \hat{f}_k(j) = 0$, and hence $e_{k+1}(j) = 0$. Thus the multiple fixed points of the ML algorithm $\left\{\hat{f}^*(j)\right\}$ (on the N^2-dimensional space for the estimate $\hat{f}$) are mapped into the unique equilibrium point $\left\{e(j) = 0\right\}$, which is the origin of the N^2-dimensional space for the iteration error. Hence the convergence rate for the restoration can be studied in terms of the trajectory behavior of the "error system" given by Equation (34) near this equilibrium point.

For small values of $e_k(j)$ (*i.e.*, near convergence to this equilibrium), $\Psi\left(e_k(j)\right)$ can be linearized by a Taylor series in the form

$$\Psi\left(e_k(j)\right) = Me_k(j) + [\text{ Higher Order Terms }] \tag{35}$$

where M is the $N^2 \times N^2$ Jacobian matrix $M = \partial\Psi / \partial e_k$. Ignoring the Higher Order Terms in the expansion, the iteration error dynamics can now be approximated by the linear system

$$e_{k+1}(j) = Me_k(j). \tag{36}$$

Solving this system of equations yields the trajectory behavior on the N^2-dimensional error space which can be described by

$$e_k(j) = M^k e_0(j) \tag{37}$$

where M^k is the state transition matrix of the system and $e_0(j)$ specifies the starting error.

From the convergent nature of the ML algorithm it follows that all eigenvalues of M are inside the unit circle, *i.e.*, $|\lambda_i(M)| < 1$ where $\lambda_i(M)$, i = 1,2, ..., N^2, denote the eigenvalues of M. Consequently, the rate of decay of the error $e_k(j)$ is larger during the first few iterations of the algorithm and progressively flattens as the number of iterations grow, a fact that is confirmed by simulation experiments as well (see Fig. 6).

3.4 Determination of Sensor Point Spread Function

It is evident from the updating rule given by Equation (24) that a knowledge of $h(x,y)$, the sensor PSF, is essential for its implementation. Indeed, for any given sensor, the accuracy with which $h(x,y)$ can be modeled determines the performance of any deconvolution algorithm used to reverse the low-pass filtering effects of the sensor and achieve a desired quality of restoration.

If the sensor is fully characterized and the imaging conditions are fully known, one may attempt to model the PSF exactly and utilize it for restoration. Imaging systems that use incoherent illumination obey the intensity convolution integral, which in the frequency domain takes the form

$$G(\omega_x, \omega_y) = H(\omega_x, \omega_y) F(\omega_x, \omega_y) \tag{38}$$

where

$$H(\omega_x, \omega_y) = \frac{\iint_{\omega_x', \omega_y'} P(\omega_x', \omega_y') P^*(\omega_x' - \omega_x, \omega_y' - \omega_y) d\omega_x' d\omega_y'}{\iint_{\omega_x', \omega_y'} \left| P(\omega_x', \omega_y') \right|^2 d\omega_x' d\omega_y'} \tag{39}$$

In Eq. (39), $P(\omega_x, \omega_y)$ is the pupil function defined by the geometry of the sensor. This equation states that $H(\omega_x, \omega_y)$, also called the Optical Transfer Function (OTF) of the imaging system, is defined by the autocorrelation of its pupil function.

In the case of a circular pupil, whose pupil function is:

$$P(\omega_x, \omega_y) = \begin{cases} 1 & , \quad \omega \leq \omega_c \\ 0 & , \quad \omega > \omega_c \end{cases} \tag{40}$$

where $\omega = \sqrt{\omega_x + \omega_y}$ and ω_c is the sensor cutoff frequency, Equation (39) can be simplified into

$$H(\omega_x, \omega_y) = \begin{cases} \frac{2}{\pi} \left[\arccos\left(\frac{\omega}{\omega_c}\right) - \frac{\omega}{\omega_c} \sqrt{1 - \left(\frac{\omega}{\omega_c}\right)^2} \right] & , \quad \omega \leq \omega_c \\ 0 & , \quad \omega > \omega_c \end{cases} . \tag{41}$$

The cutoff frequency ω_c can be related to the sensor parameters by the relation

$$\omega_c = \frac{2\pi D_p}{\lambda f} \tag{42}$$

where D_p is the diameter of the circular pupil, λ is the wavelength of the incoming radiation, and f is the focal length. It may be noted that for a sensor with f/1 optics, ω_c can be determined simply from $\omega_c = \frac{2\pi}{\lambda}$. Once the OTF is determined from Equation (41), the PSF can be readily computed by taking its inverse Fourier transform.

In practice, most incoherent imaging systems sample the focal plane image on a discrete grid of samples. The sampling in the image plane can be achieved either with a fixed array (fully staring array) or by scanning, either mechanically or by electronic means, of a small number of detectors across the image. The expression for ω_c, given by Equation (42), is useful for determining the spacing between the detectors and/or for obtaining a scan rate for collection of samples.

While the expressions in Equations (41) and (42) help determine the PSF from using the sensor parameters, the PSF can also be experimentally determined for a given sensor by attempting to image a point source or an edge source. A commonly used approximation to an ideal point source is to employ a Gunn diode oscillator (GDO) source whose separation from the sensor aperture can be adjusted to appear as a point source [27]. The distribution at the focal plane can then be used to model the PSF of the sensor. Even in cases where a complete characterization of the sensor and the imaging conditions are not available, one may conduct an analysis of the acquired image data to approximately identify the cutoff frequency ω_c [28]. Starting with an intensity profile corresponding to an edge of the object in the acquired image, an OTF can be created as a low pass filter function with an adjustable cutoff frequency whose convolution with an input pulse object yields an outcome which matches the edge profile selected. While this process gives only a rough estimate of the OTF, by an iterative adjustment of this estimate to implement a blind deconvolution scheme [24], a more accurate OTF can be developed. Such a procedure is of considerable practical usefulness since even if the sensor parameters are accurately known, during the operation several external factors such as vibration of the sensor platform, random atmospheric turbulence, blur due to target motion and blur due to out-of-focus imaging can change the overall PSF of the imaging process. In this

case, the PSF determined from using the known sensor parameter values can be used as the initial estimate that can be iteratively refined through blind deconvolution adjustment steps as outlined in the next section.

3.5 Analysis of Processor Requirements for Implementation of ML Algorithm

As noted earlier, iterative approaches to image restoration possess considerable advantages over non-iterative restoration procedures which attempt to implement an inversion of the image formation process described by Equation (1) in one step. Despite these benefits, there are several claims in the literature [29] that iterative algorithms demand excessive computational resources not conducive for practical implementations and one should settle for a noniterative implementation even at the cost of inferior restoration and/or super-resolution performance. The aim of the present analysis is to provide some quantitative data that demonstrate viable implementations of the iterative ML algorithm given by Equation (24) with available microprocessors. In the comparison of iterative and non-iterative algorithms for super-resolution, it should be kept in mind that iterative approaches provide the flexibility for applying the constraints that are fundamental to achieving the required spectral extrapolation in a distributed fashion as the solution progresses, and hence the computations at each iteration are generally far less intensive than the single-step computation of non-iterative approaches [28].

For evaluating the maximum processing time each iteration of the algorithm may demand, an examination of the mathematical operations performed in Equation (24) indicates that there are two 2-dimensional (2-D) fast Fourier transforms (FFTs), two 2-D Inverse fast Fourier transforms (IFFTs), three multiplications (point-by-point) and one division (point-by-point) performed. An approximate upper bound on the total computational load can hence be projected as a requirement for six 2-D FFTs in each ML iteration (by bounding the two multiplications, one division and any other needed operations (for thresholding, scaling, formatting, etc.) by a rather generous limit of two 2-D FFTs).

For estimating the processing time, and the processing capacity in image frames/sec, one may use a typical illustrative microprocessor implementation based on TMS320C6x DSP chip for which data is readily available [30]. For a *N*x*N* image, the total number of cycles for implementing a 2-D FFT is given by

$$N_C = 2N\left[(\log_4 N)\times\left(10\times\frac{N}{4}+33\right)+7+\frac{N}{4}\right] \tag{43}$$

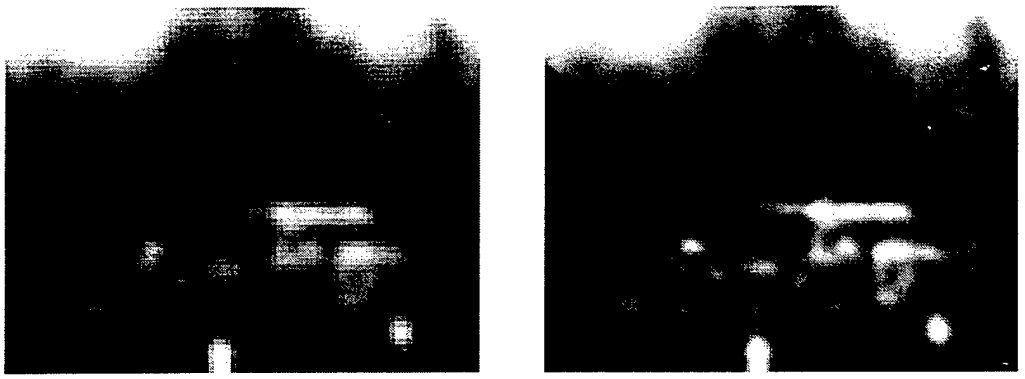
and hence an upper bound on the number of cycles for implementing two ML iterations can be obtained as

$$N_{2ML} \le 24N\left[(\log_4 N)\times\left(10\times\frac{N}{4}+33\right)+7+\frac{N}{4}\right] \tag{44}$$

The motivation for using two ML iterations in the above computation comes from the observation that the convergence of the restoration error in statistical restoration algorithms in general, and ML restoration in particular, is rather very steep during the first few iterations. In fact, in most practical implementations where the goal of processing is super-resolution, one would notice a major amount of spectral extrapolation (and corresponding resolution improvements) occurring during the first one or two iterations with succeeding iterations adding progressively lesser amounts of spectral components, Consequently, in applications where the processing time is of critical importance, one may stop the iterations on an image frame after the first two and move to the next frame.

Fig. 10a shows a PMMW image ("Humvee image") of size 84x64 on which processing experiments were conducted to evaluate the resolution enhancements resulting at the end of two ML iterations. This image was supplied to us by the Wright Laboratory Armament Directorate. Only a minimal information about the sensor that the image was recorded by a single detector radiometer with 1ft diameter aperture at 95 GHz was available. The processed image at the end of two ML iterations is shown in Fig. 10b where the enhanced edge structure of the vehicle and the improved resolution for identifying the details are clearly evident. Since a complete characterization of the sensor and the imaging conditions were not available, an approximate analysis was conducted to obtain the PSF for the above processing. Starting with an intensity profile corresponding to a selected edge of the object in the image, a PSF was

created to simulate a low-pass filter whose convolution with an input rectangular pulse yields an outcome which is similar to the edge profile.



Figs. 10a and 10b. Acquired PMMW image and its restoration after 2 ML iterations

Table 1 gives quantitative data for different image sizes (N) of N_{2ML} as well as the time in seconds for implementing two ML iterations and the corresponding processing capacity in number of frames processed per second. It is instructive to note that a processing rate that exceeds 50 frames of size 128x128 per second (a rate that exceeds video rates) is supported by this commercially available chip implementation. With additional efforts at optimizing the

N	Number of Cycles N_{2ML}	Time (in sec) for 2 ML iterations	Processing Capacity (Number of frames/sec)
512	74,262,528	0.3713	2.69
256	16,975,872	0.0848	11.79
128	3,915,264	0.0195	51.08

Table 1. Typical processing capacities for iterative ML super-resolution.

implementation, by limiting the processing to only a smaller region of interest in the image for instance, further increases in processing capacity can be achieved. Thus, based on the above analysis, we conclude that practical implementations of iterative ML super-resolution algorithms of the type discussed here are quite realistic even in applications where processing time is of critical importance and hence one need not settle for non-iterative processing algorithms that may not provide all of the advantages (robustness, quality of restoration, suppression of noise effects, etc.) that well-tailored iterative ML algorithms are capable of providing.

3.6 Super-resolution of PMMW Images

In this section we shall outline the results of a number of processing experiments that were conducted to assess the capability of the ML algorithm described by Equation (24) in the restoration and super-resolution of PMMW images. During the course o this project we worked very closely with a number of research groups that were developing state-of-the-art PMMW radiometers. Consequently, we were able to obtain a number of PMMW images acquired from different sensor designs and collected under different conditions which has resulted in a large database of PMMW imagery data available in our research laboratory (Information Processing & Decision Systems Lab) at the University of Arizona. The data-sets that we specifically used for processing with the ML algorithm came from three camera designs: (i) a single detector 95 GHz radiometer with 1 *ft* aperture built by Intelligent Machine Technology (IMT) corporation for the Air Force Research Laboratories (AFRL), (ii) a PMMW camera built by TRW Inc. that operates at 90 GHz with a 17 Hz framerate, and (iii) a 95 GHz sensor built by the Army Research Laboratories (ARL) with a 3 *ft* parabolic antenna. It may be mentioned that while the images from the TRW camera were captured from an airborne platform when the camera was flown on a UH-1N helicopter over Camp Pendleton in California and the images were acquired from altitudes between 500 and 2000 *ft* at a viewing angle of 45 *deg* from nadir, the images from the ARL and the AFRL/IMT radiometers were acquired from a ground-based platform. Furthermore, these images were collected at different ranges and during different times of day thus providing a non-uniform data-set for testing the efficiency of the algorithm.

Fig. 11a shows the image of a tank (a950.sdt) acquired by the TRW camera from an altitude of 800 ft. The poor resolution in the image is evident and the features of the object are not identifiable. In Fig.11b is shown the processed image after 12 iterations of the ML algorithm, which clearly demonstrates the enhancement of resolution. For the sake of displaying all the features of the tank that were brought up by super-resolution processing, the processed image is also shown in Fig. 11c after an additional step of histogram equalization. This last step merely adjusts the image contrast and does not lead to any further resolution enhancement, and hence does not require any significant computation. One can easily

appreciate the restoration of the original image and the benefits of iterative processing for recognition and classification of the object.

Fig. 11a. Acquired image (a950.sdt)

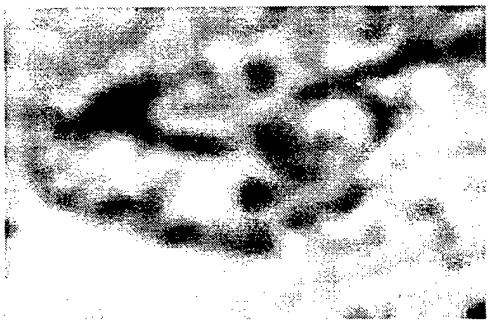


Fig. 11b. ML Processed Image (after 12 iterations)

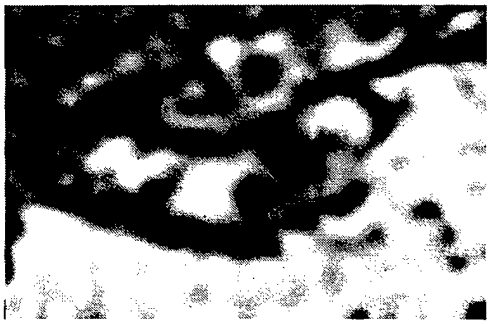


Fig. 11c. ML processed Image (after 12 iterations and histogram equalization)

Fig.11. Results of processing "Tank Image" (a950.sdt)

In another experiment, we processed the PMMW image (c10869.sdt) shown in Fig. 12a which was also acquired by the TRW camera. This is the image of a runway with some buildings on the side and was taken from an altitude of 2000 ft. Fig. 12b shows the processed image obtained after 10 iterations of the ML algorithm. The same image with the additional step of histogram equalization is shown in Fig. 12c. Once again, the resolution improvements obtained from super-resolution processing are clearly evident.

Fig. 12a. Acquired image (c10869.sdt)

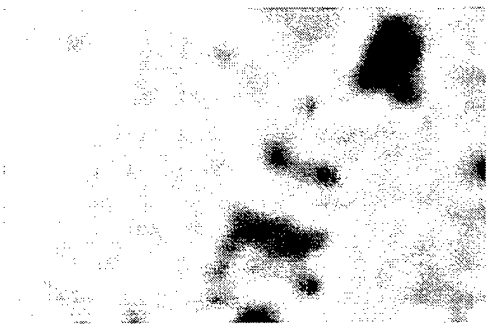


Fig. 12b. ML Processed Image (after 10 iterations)

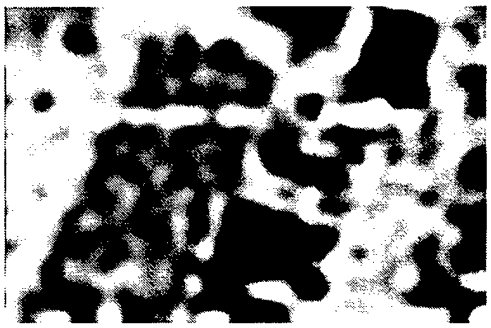


Fig. 12c. ML processed Image (after 10 iterations and histogram equalization)

Fig. 12. Results of processing "Runway and Buildings Image" (c10869.sdt)

Fig. 13a shows the image of a tank (t12mod12.tif) acquired by the AFRL/IMT radiometer. The processed image after 25 iterations of the ML algorithm is shown in Fig. 13b and the same image with additional histogram equalization is shown in Fig. 13c. To display the additional enhancement in resolution, the processing was continued for 100 iterations. Fig. 13d shows the resulting image at the end of 100 iterations and the same image with additional histogram equalization is shown in Fig. 13e. The progressive improvements in the tank features particularly at the gun barrel and the wheels as well as in the background (the tree branches) are to be noted.

Fig. 13a. "Tank" image (t12mod12.tif) acquired by AFRL/IMT radiometer

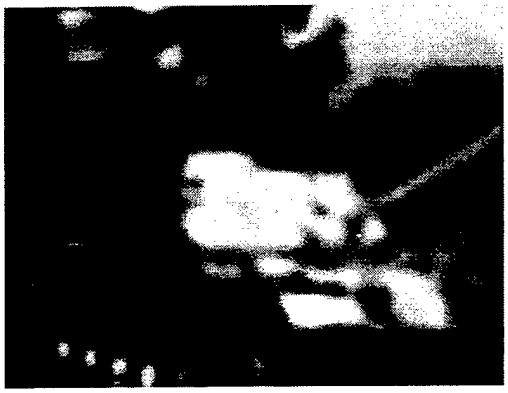


Fig. 13b. ML Processed image (after 25 iterations)

Fig. 13c. ML Processed image (after 25 iterations and histogram equalization)

Fig. 13d. ML Processed image (after 100 iterations)

Fig. 13e. ML Processed image (after 100 iterations and histogram equalization)

Fig. 13. Results of processing AFRL "Tank Image" (t12mod12.tif)

Another PMMW image of a building "La Quinta Inn" (t12mod12.tif) acquired from the same AFRL/IMT camera is shown in Fig. 14a. The processed image after 25 iterations of the

ML algorithm is shown in Fig. 14b and the same image with additional histogram equalization is shown in Fig. 14c. The image obtained by continuing the processing for 100 iterations is shown in Fig. 14d and this image with a further histogram equalization is shown in Fig. 14e.

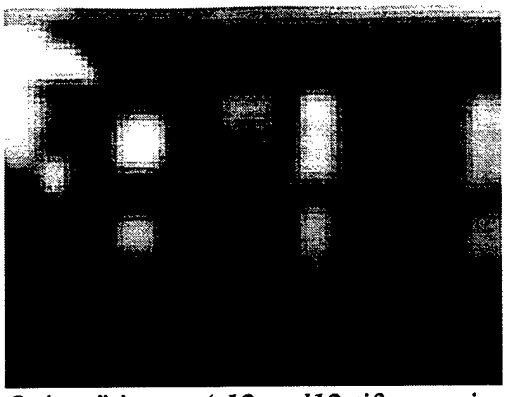


Fig. 14a. "La Quinta" image (t12mod12.tif) acquired by AFRL/IMT radiometer

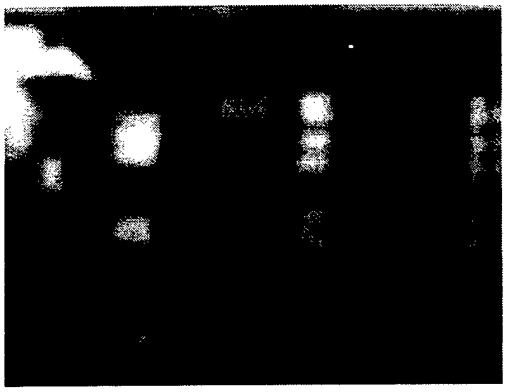


Fig. 14b. ML Processed image (after 25 iterations)

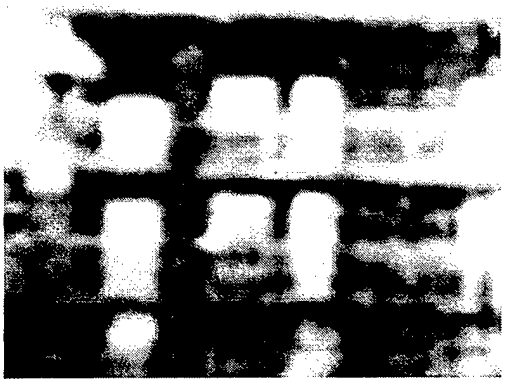


Fig. 14c. ML Processed image (after 25 iterations and histogram equalization)

Fig. 14d. ML Processed image (after 100 iterations)

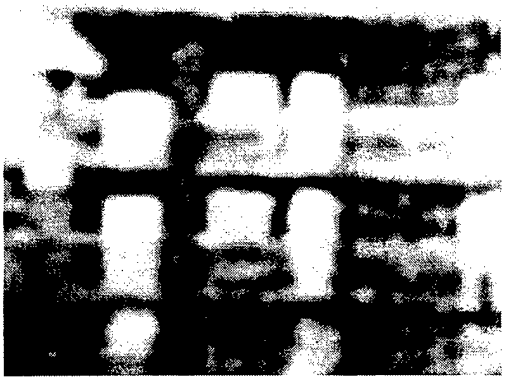
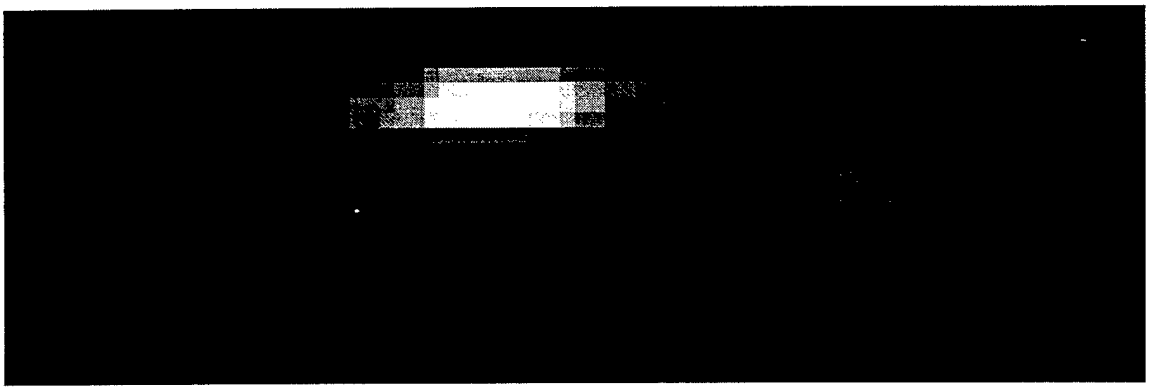


Fig. 14e. ML Processed image (after 100 iterations and histogram equalization)

Fig.14. Results of processing AFRL " La Quinta Inn" image (t12mod12.tif)

In yet another experiment we processed two images of a Bradley tank (Brad45a.bmp and Brad90a.bmp) acquired by the ARL radiometer from two different angles (45 and 90 degrees) at a range of 43 *meters* and a scan step size of 0.1 *deg*. These images as well as the processed images after 10 iterations of the algorithm are shown in Figs. 15 a-b and 16 a-b. The improvement in the quality of the images is clearly evident. For the sake of displaying all the features of the tank that were brought up by super-resolution processing, the processed images are also shown in Figs. 15c and 16c with a reversed intensity scale that interchanges the bright and the dark regions of the image.



. Fig. 15a. Acquired Image (Brad45a.bmp)

Fig. 15b. ML Processed Image (after 10 iterations)

Fig. 15c. ML Processed Image (with intensity reversal)

Figs. 15a-c. Results of processing "Bradley Tank" image (Brad45a.bmp)

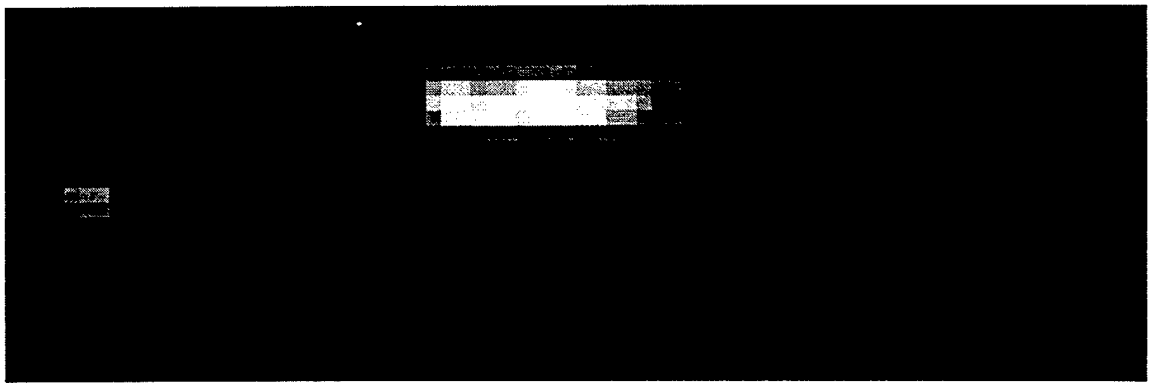


Fig. 16a. Acquired Image (Brad90a.bmp)

Fig. 16b. ML Processed Image (after 10 iterations)

Fig. 16c. ML Processed Image (with intensity reversal)

Figs. 16a-c. Results of processing "Bradley Tank" image (Brad90a.bmp)

4. MODIFIED VERSIONS OF ML ALGORITHM

While the performance of the ML algorithm for restoring and super-resolving degraded images is quite appealing as demonstrated in the last section, in practical applications there could be certain non-ideal conditions, such as imperfect knowledge of the PSF of sensor and poor Signal-to-Noise (SNR) in the image to be processed, which may result in a suboptimal performance. Appropriate modifications may need to be used to restore optimal performance in these cases. In this section we shall discuss a few scenarios where some modifications to the ML algorithm could be developed with an objective of obtaining improved performance.

4.1 Robustness to PSF Uncertainties and a Blind ML Algorithm

It is generally well known that the quality of restoration depends critically on the extent of knowledge of the sensor PSF. If the sensor is fully characterized and the imaging conditions are fully known, one may attempt to model the PSF exactly and utilize it for restoration as discussed previously. Unfortunately, in a real operation environment such as that encountered by a missile in flight, several external factors such as the vibration of the imaging system, random atmospheric turbulence, blur due to target motion, blur due to out-of-focus imaging etc., can change the overall PSF of the imaging system. These factors, together with any problems associated with poor calibration of the deployed sensor, can make any assumptions of perfect knowledge of PSF not entirely realistic to use in practice. Consequently, a certain degree of robustness to uncertainties in estimating the PSF parameters is useful.

To appreciate the importance of this feature in restoration operations, it suffices to consider a trivial restoration exercise in a simple noise-free scenario. Let $f = \begin{bmatrix} 1 & 0 \end{bmatrix}^T$ be a simple object vector imaged by a sensor characterized by a PSF matrix $H = \begin{bmatrix} 1 & 0.9 \\ 0.9 & 1 \end{bmatrix}$. Then, with $n = 0$ in Equation (3), the image obtained is $g = Hf = \begin{bmatrix} 1 & 0.9 \end{bmatrix}^T$. The closeness of the two elements in g, compared to those of f, illustrates the blurring caused by the sensor. If an exact knowledge of H is available, one may compute $H^{-1} = \frac{1}{0.19}\begin{bmatrix} 1 & -0.9 \\ -0.9 & 1 \end{bmatrix}$ and restore the

object by implementing $\hat{f} = H^{-1}g$ which yields $\hat{f} = \begin{bmatrix} 1 & 0 \end{bmatrix}^T$. However, if the sensor PSF is modeled even with a small uncertainty, very drastic errors in object estimates could result. To illustrate, consider that the actual PSF of the sensor is $H_{actual} = \begin{bmatrix} 1.1 & 0.8 \\ 0.8 & 1.1 \end{bmatrix}$, which results in the image $g = H_{actual} f = \begin{bmatrix} 1.1 & 0.8 \end{bmatrix}^T$. It may be noted that the variation in the values of the image elements is rather small. However, with the sensor being modeled with matrix H, the use of H^{-1} in image restoration yields the estimate $\hat{f} = H^{-1}g = \begin{bmatrix} 2 & -1 \end{bmatrix}^T$, which signifies a complete breakdown in the restoration process.

The example cited above illustrates the significance of uncertainties in PSF parameters even in noise-free restorations. The impact of these uncertainties attains a considerably greater magnitude when significant noise levels are to be taken into account. The interplay between noise and PSF errors can be seen simply from the convolutional model given by Equation (2). Let $h(i)$, the estimated PSF, contain an error denoted by $\Delta h(i)$, *i.e.* the true PSF of the sensor $h_{true}(i) = h(i) - \Delta h(i)$. Then,

$$g(i) = \sum_{j=1}^{N} \left[h_{true}(i-j) + \Delta h(i-j) \right] f(j) + n(i)$$

$$= \sum_{j=1}^{N} h_{true}(i-j) f(j) + \left[n(i) + \sum_{j=1}^{N} \Delta h(i-j) f(j) \right], \tag{45}$$

which indicates a change in the noise level from $n(i)$ to $\tilde{n}(i) = n(i) + \sum_{j=1}^{N} \Delta h(i-j) f(j)$ thus accentuating its effects.

A super-resolution algorithm that constructs object estimates in an iterative fashion by starting with an approximately estimated PSF of the sensor and progressively modifies the shape of the PSF (or equivalently, its Fourier transform, which is the Optical Transfer Function (OTF)) as part of the iterative procedure can be designed using updating rules similar to that in Equation (24). This algorithm is developed using a statistical framework for tailoring the implementation rules and aims at maximizing the likelihood function relating the image

data to the object estimate at any given stage of the process. Since a joint estimation of both object and the sensor PSF is performed, this algorithm performs a blind restoration of the input image. Hence the algorithm is ideal to use when the initial PSF estimate is known to be inaccurate or an approximate PSF constructed from analysis of the image available is all that can be had to commence the restoration process.

The algorithm iteratively implements several cycles of recursive estimation. In each cycle of implementation, there are two distinct steps that will be executed which are briefly outlined in the following:

Step 1: With the initial object estimate $\hat{f}_o(j) = g(j)$ and the available PSF estimate $\hat{h}_l(j)$, implement m iterations of object estimation by

$$\hat{f}_{k+1}(j) = \hat{f}_k(j)\left[\left\{\frac{g(j)}{\hat{f}_k(j) \otimes h_l(j)}\right\} \otimes h_l(j)\right], \quad k = 0,1,2,\ldots,(m\text{-}1); \quad j = 1,2,\ldots,N, \tag{46}$$

where k denotes the iteration count and $\otimes$ denotes discrete convolution, as before.

Step 2: With the object estimate $\hat{f}_m(j)$ obtained at the end of Step 1, implement n iterations of PSF updating by

$$h_{l+1}(j) = h_l(j)\left[\left\{\frac{g(j)}{h_l(j) \otimes \hat{f}_m(j)}\right\} \otimes \hat{f}_m(j)\right], \quad l = 0,1,2,\ldots,(m\text{-}1); \quad j = 1,2,\ldots,N, \tag{47}$$

where l denotes the iteration count.

A flow-chart depicting the iterative implementation of the two steps described above is shown in Fig. 17. The algorithm can be run iteratively over several cycles until a specified maximum iteration count is reached or a processed image with a satisfactory resolution level is attained.

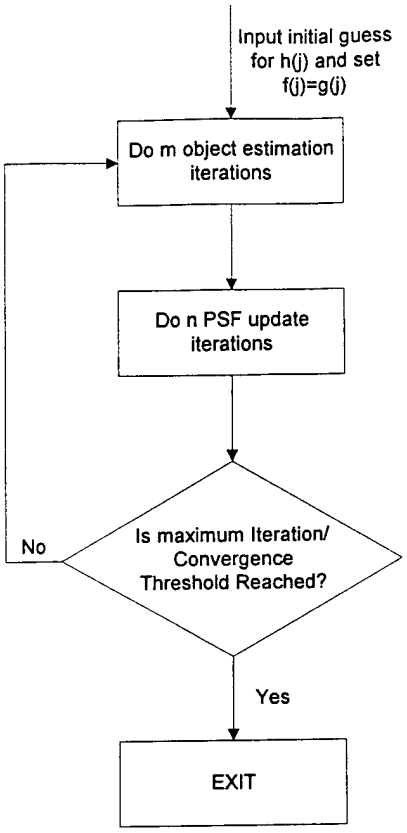


Fig. 17. Flow chart for implementation of Blind ML Algorithm.

The super-resolution performance of this algorithm has been tested through several experiments conducted with both one- and two-dimensional signals (including in particular some PMMW images obtained with rather poor spatial resolution). In the following, we shall demonstrate the robustness of the algorithm to variations in the starting estimates of PSF parameters. An insight into this attractive property of robustness can be given from the underlying ML estimation procedure employed. Since, at the most fundamental level, an ML estimation algorithm relates conditional probabilities through statistical arguments, it possesses intrinsic capabilities to account for statistical fluctuations in the signals and the imaging process.

In order to conduct a controlled experiment where the errors in signal restoration (in spatial domain) and in PSF estimation (or equivalently in the estimation of OTF, measured in the frequency domain) could be monitored over several cycles of algorithm implementation, an

experiment was conducted starting with a one-dimensional object consisting of three pulses of uneven heights, shown in Fig. 18a. To obtain an image to be used as input to the super-resolution algorithm, a noisy blurred image was constructed by convolving the object with a sensor with cutoff frequency $\omega_c = 30$ and a OTF with a triangular profile whose magnitude varies linearly from a normalized value 1 at $\omega = 0$ to 0 at $\omega = \omega_c = 30$ (shown in Fig. 18b), and by adding Gaussian noise to produce a signal with SNR = 14.9391 dB. The resulting blurred signal, shown in Fig. 18c, is processed by the blind ML restoration algorithm. Since our present objective is to demonstrate the robustness of the algorithm to inaccurate selections of PSF parameters (or equivalently OTF), two implementations of the algorithm were made with two distinct OTF profiles, one with a cutoff frequency 28 (hereafter referred to as *Implementation 1*) and the other with a larger error made by selecting a cutoff frequency 25 (referred to as *Implementation 2*). In each implementation, the algorithm was run through 10 cycles with each cycle comprising of five object estimation iterations (i.e. $m = 5$ in Step 1) followed by two PSF updating iterations (i.e. $n = 2$ in Step 2).

Fig. 18d shows the initial OTF profile for Implementation 1 (note the cutoff at 28). The restored object at the end of the 10^{th} cycle is shown in Fig. 18e and the reshaped OTF is shown in Fig. 3f for this implementation. It is evident that the peaks are better resolved, and more significantly, the frequency range is extended attesting to the super-resolution capabilities of the algorithm. The corresponding performance results for Implementation 2 are shown in Figs. 18g, 18h and 18i, which display the initial selection of OTF profile (with cutoff at 25), the restored object, and the reshaped OTF, respectively. The robustness of the algorithm is evident from the fact that the performance has not degraded much despite the larger error made in the selection of OTF. A greater insight into the robustness feature of the algorithm can also be obtained from Figs. 18j and 18k, which plot the variation in estimation errors (mean square error in object estimation and in OTF estimation) as the number of cycles of algorithm implementation is increased.

Fig. 18a

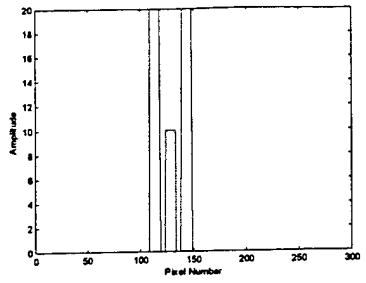


Fig. 18b

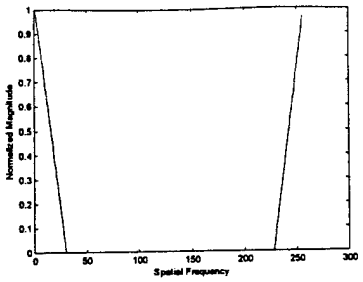


Fig. 18c

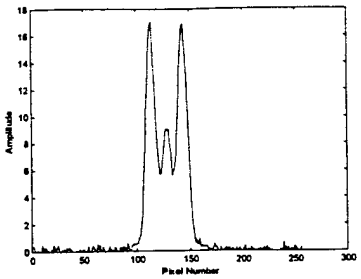


Fig. 18d

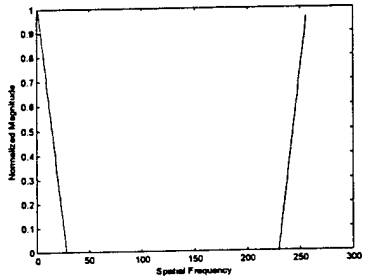


Fig. 18e

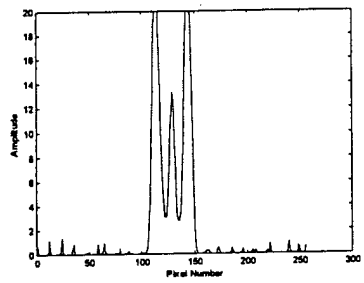


Fig. 18f

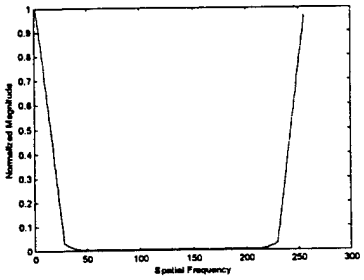


Fig. 18g

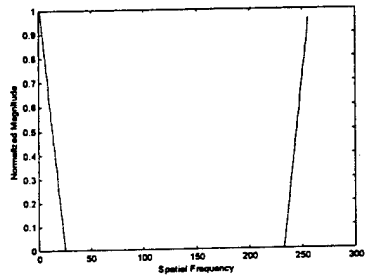


Fig. 18h

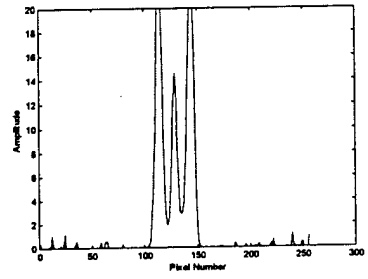


Fig. 18i

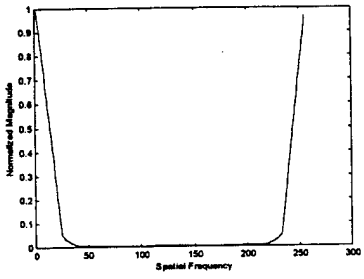


Fig. 18j

Fig. 18k

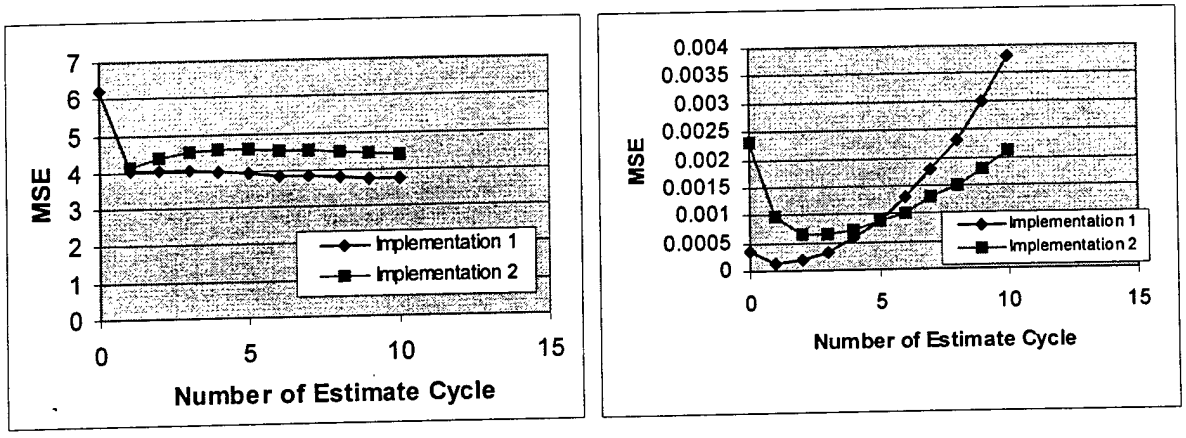


Fig. 18. Demonstration of robustness of Blind ML Algorithm

We shall now describe the results of two experiments in order to demonstrate the performance of the blind ML restoration algorithm in estimating both the object an the sensor PSF simultaneously. In the first experiment, we will consider a simulated one-dimensional object while in the second experiment we will consider the super-resolution of a real PMMW image.

Starting with a one-dimensional object consisting of three pulses of uneven heights, shown in Fig. 19a, a blurred image, shown in Fig. 19b, was obtained by convolving with a sensor with cutoff frequency 14 and a OTF whose profile is shown in Fig. 19c. For commencing the blind ML iterations, an initial estimate of PSF was made by assuming an OTF with cutoff frequency 10 (an error was made deliberately to test the performance of the algorithm to yield estimates progressively moving towards the true cutoff frequency of 17) and taking its inverse Fourier transform. The assumed OTF and the PSF are shown in Figs. 19d and 19e.

The OTF estimates resulting after 1 and 3 cycles of the algorithm are shown in Figs. 19f and 19g which indicates the progression of the algorithm in expanding the OTF towards the true cutoff frequency. Each cycle of algorithm implementation consists of five object estimation iterations ($m = 5$) followed by two PSF updating iterations ($n = 2$). The final results after 10 cycles of algorithm implementation are shown in Figs. 19h, 19i and 19j, which show the estimated OTF, the corresponding PSF and the restored object respectively.

In the second experiment we tested the performance of the blind ML restoration algorithm in processing PMMW images supplied by the Wright Laboratory Armament Directorate. The specific image used as input to the algorithm is the "Parking Lot 3" image which is shown in Fig. 20a. For obtaining an initial estimate of the PSF to commence the iterations, an analysis similar to the one described earlier of matching the edge profiles was made and an OTF with a cutoff frequency of 82, shown in Fig. 20b (only one dimension of OTF is shown here, for simplicity), was developed. The ML restoration was implemented over 20 cycles with 5 ML estimation iterations ($m = 5$) and 2 PSF updating iterations ($n = 2$) in each cycle. The restored images at the end of 5 cycles and 20 cycles are shown in Figs. 20c

Fig. 19a

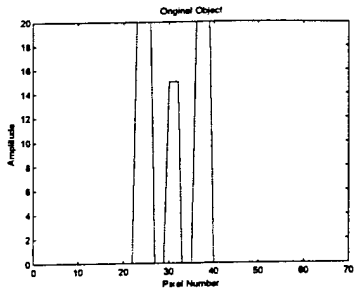


Fig. 19b

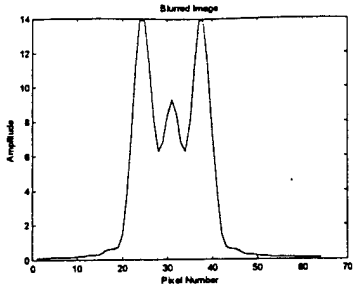


Fig. 19c

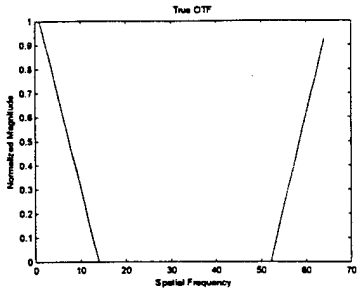


Fig. 19d

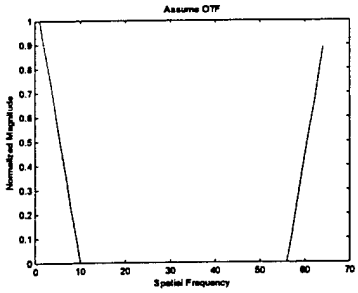


Fig. 19e

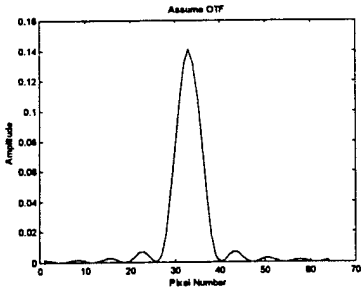


Fig. 19f

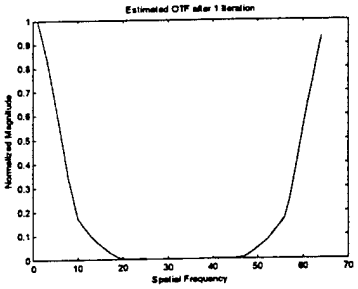


Fig. 19g

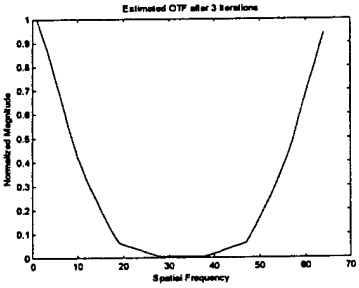


Fig. 19h

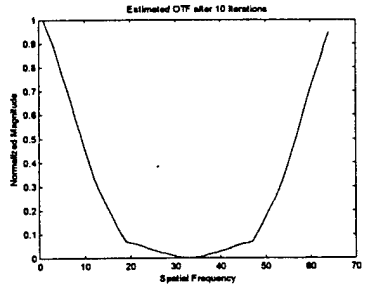


Fig. 19i

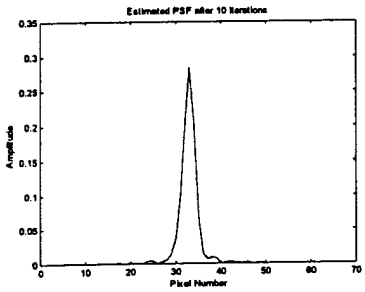


Fig. 19j

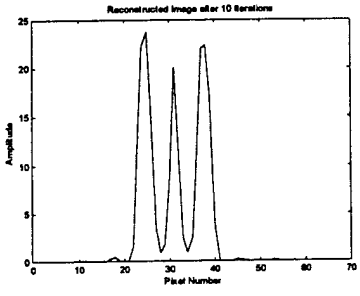


Figure 19. Joint estimation of object and PSF by blind ML restoration

and 20d. The reshaped OTF at the end of 20 cycles is also shown in Fig. 20e. The progressive enhancement of resolution is clearly evident from the improved structural details of the building and the parked automobiles.

Fig. 20a

Fig. 20b

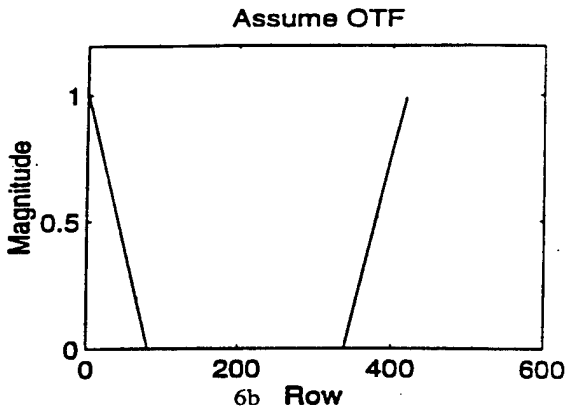


Fig. 20c

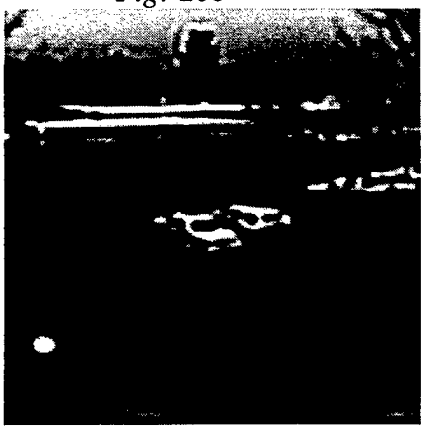


Fig. 20d

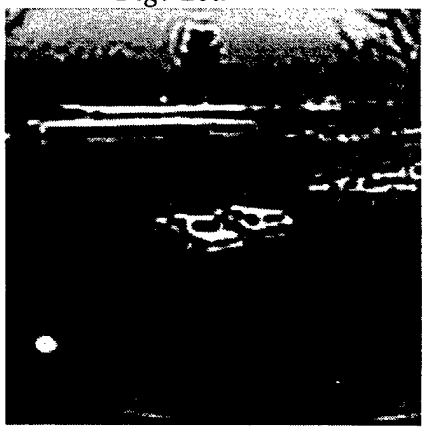


Fig. 20e

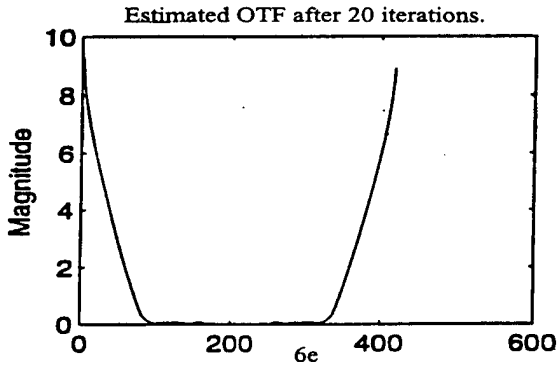


Figure 20. Blind ML restoration of PMMW image ("Parking Lot 3").

4.2 Suppression of Noise Induced Artifacts

It is well known that image restoration algorithms generally produce undesirable artifacts because of the ill-posed nature of the deconvolution problem. As noted in the Introduction, different types of artifacts are generated during the restoration process and these are due to various causes. In the development of super-resolution algorithms for applications in missile seekers, we are particularly interested in noise induced artifacts due to reasons that will be elaborated in the following discussion.

First of all, it should be emphasized that artifacts generated due to the presence of noise in the image data manifest themselves in the restored image much differently from the more familiar "ringing artifacts" [18]. The latter are primarily due to Gibbs phenomena resulting from the truncation of the Fourier components caused by the extent of the acquired spatial frequencies of the scene or object imaged being finite. Consequently, during the restoration process, particularly when ML estimation methods are employed, noticeable distortions near edges occur as sharp transitions, or edges that may be present in the intensity distribution function become more accentuated and emerge with overshoots (or ringing). For our present applications, where the goal of restoration is to assist in feature extraction operations, ringing artifacts which essentially show up as periodic (or decaying periodic) repetitions of sharp intensity transitions in the image may not be objectionable. However, noise induced artifacts, which could show up with more general patterns, can be quite troublesome since they may obscure some critical feature of the target, introduce features that may not be present in the scene, or otherwise cause distortions of the feature maps that will be extracted from the restored image.

In super-resolution processing, where the primary goal is the creation of new frequencies which are present in the object imaged but not in the image recorded, the artifacts resulting from the presence of high frequency noise need to be given special consideration. Due to the spectral mixing that takes place during the execution of each iteration, it is likely that the high frequency components of the noise become responsible for the expansion of the

image bandwidth, which show up as artifacts in the processed image. Suppression of such noise induced artifacts is hence a desirable step in the overall processing.

For gaining insight into how the presence of noise affects the performance of the blind ML restoration algorithm discussed in the previous section, a controlled experiment was conducted with a one-dimensional object blurred with a known PSF function and restoration conducted in a noise-free scenario as well as when a significant amount of noise is added. Fig. 21a shows the object with three spikes that was blurred with a OTF whose profile is shown in Fig. 21b. The resulting blurred image is shown in Fig. 21c. Processing of this image by the blind ML restoration algorithm with a starting selection of OTF shown in Fig. 21d results in the restored image at the end of 10 cycles of algorithm implementation (with $m = 5$ and $n = 2$) shown in Fig. 21e, and the reshaped OTF at the end of 10 cycles is shown in Fig. 21f. The experiment was repeated with the addition of Gaussian noise creating a blurred image with SNR = 11.2494 dB shown in Fig. 21g. Processing of this image with the same starting selection of OTF as before (*i.e.* OTF shown in Fig. 21d) resulted in the restored image shown in Fig. 21h at the end of 10 cycles, and a reshaped OTF shown in Fig. 21i.

The significant errors introduced in OTF estimation in the processing of the noisy image are particularly noteworthy and illustrate the effects of the added noise. Evidently, some form of filtering should be used to mitigate these effects and obtain satisfactory restoration performance. Two choices are available: (i) *pre-filtering* of the noisy image prior to restoration processing, and (ii) *post-filtering* of the restored image to correct for the degradations caused by noise. The latter appears to be a more feasible option keeping in mind the goals of achieving improved resolution in missile seeker applications, since pre-filtering the image while successful in reducing the noise contamination also reduces the resolution in the image. In restoration studies, it is generally known that pre-filtering methods aimed at artifact reduction extract a price in yielding reduced resolution levels in the processed image.

Fig. 21a

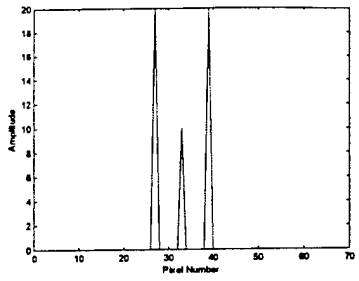


Fig. 21b

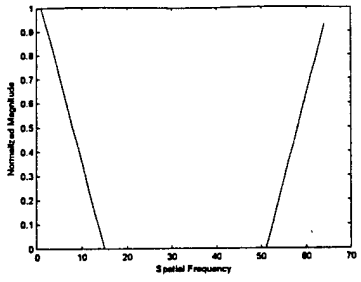


Fig. 21c

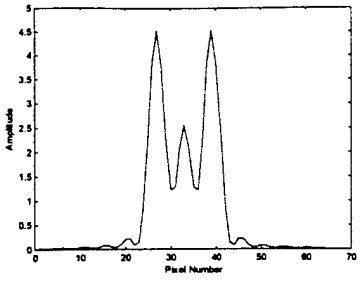


Fig. 21d

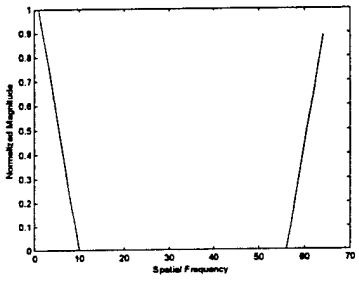


Fig. 21e

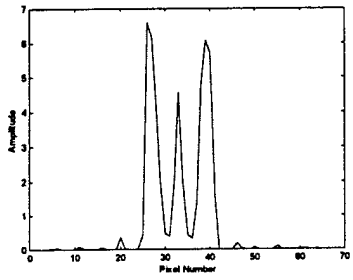


Fig. 21f

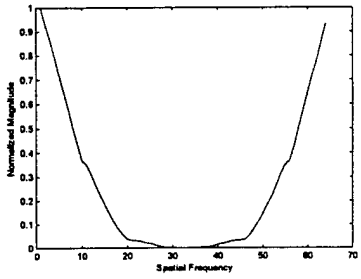


Fig. 21g

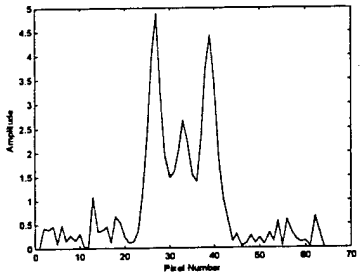


Fig. 21h

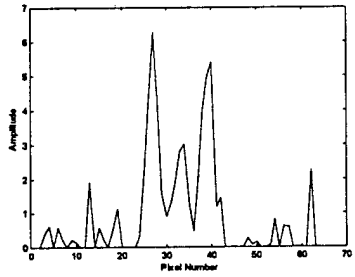


Fig. 21i

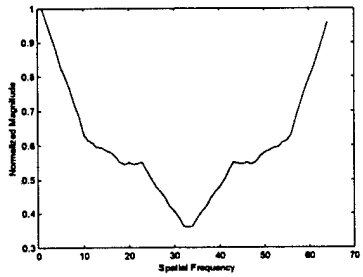


Fig. 21. Results of processing one-dimensional signal with three spikes (controlled experiment)

In the following, we shall briefly outline two post-filtering adjustments that result in modified restoration algorithms. We shall refer to these as Blind ML-PF1 and Blind ML-PF2 algorithms in further discussion. The commonality between these is to adjust the OTF function subsequent to the PSF estimation step (*i.e.* Step 2 of the blind ML restoration cycle) by utilizing the low-pass filtering characteristic of physical sensors which imposes a monotonically nonincreasing behavior on the OTF magnitude near the cutoff frequency as well

as typical restrictions on the width of passband relative to the sampling frequency. However, since the cutoff frequency is not known *a priori*, and the goal of blind ML restoration is to progressively extend the width of passband, we use an adjustment procedure suggested from empirical considerations of setting the OTF magnitude to zero for all frequencies beyond 60 % of the folding frequency after the PSF estimation in Step 2. This adjustment results in the modified algorithm Blind ML-PF1 which updates the iterative algorithm given in the previous section by modifying Step 2 as follows:

Step 2*: Using object estimate $\hat{f}_m(j)$ obtained at the end of Step 1, implement n iterations of PSF updating together with an OTF adjustment step by

$$h_{l+1}(j) = h_l(j)\left[\left\{\frac{g(j)}{h_l(j) \otimes \hat{f}_m(j)}\right\} \otimes \hat{f}_m(j)\right], j = 0,1,2,\ldots, N \tag{48}$$

and

$$\begin{aligned} H_{l+1}(\omega) &= H_{l+1}(\omega), && \omega \le 0.6 f_s/2 && l = 0,1,2,\ldots,(n\text{-}1) \\ &= 0, && \omega > 0.6 f_s/2 \end{aligned} \tag{49}$$

In Equation (49), $H_{l+1}(\omega)$ denotes the DFT of $h_{l+1}(j)$, $j = 1, 2, \ldots, N$, and ω_s denotes the sampling frequency. The effect of the adjustment given by this equation in compensating for the overextension of bandwidth due to the higher frequency components of noise is quite evident.

An alternative approach for the adjustment of OTF results in improved performance in certain cases and involves replacing the adjustment rule in Equation (49) by the following:

$$\begin{aligned} H_{l+1}(\omega) &= H_{l+1}(\omega) \quad \text{if} \quad H_{l+1}(\omega) \le H_{l+1}(\omega - 1) \\ &= 0 \quad \text{if} \quad H_{l+1}(\omega) > H_{l+1}(\omega) \end{aligned} \tag{50}$$

The rationale behind the adjustment suggested by Equation (50) is to remove the possible ripples in the OTF outside the passband contributed by the higher frequency components of noise by setting the OTF magnitude to zero if it starts increasing relative to its value at the previous frequency step. We shall hereafter refer to this algorithm as Blind ML-PF2.

In the following we shall briefly present results of some experiments to highlight the performance improvements obtained from the modified algorithms.

Experiment 1: In this experiment, we consider the restoration of the one-dimensional object with three spikes that was discussed earlier. Processing of the blurred image in the noise-free case, which is shown in Fig. 21c, with the same starting selection of OTF as before (shown in Fig. 21d) results in the restored image at the end of 10 cycles (with $m = 5$ and $n = 2$) of implementing Blind ML-PF1 algorithm which is shown in Fig. 22a. The reshaped OTF in this case is shown in Fig. 22b. The corresponding results for the implementation of Blind ML-PF2 algorithm are shown in Figs. 22c and 22d. For a more direct comparison with the performance of the original algorithm, we show in Figs. 22e and 22f the variations in the mean square error in object estimation and in OTF estimation as the iteration cycles progress for the original algorithm and for Blind ML-PF1 and Blind ML-PF2 algorithms. These sketches indicate very little changes in the estimation errors for the various algorithms in this case of processing the noise-free signal. However, considering now the restoration processing of the noisy blurred image shown in Fig. 21g (with SNR = 11.2494 dB), with the same starting selection of OTF as before, Blind ML-PF1 algorithm results in a restored image shown in Fig. 22g and a reshaped OTF shown in Fig. 22h at the end of 10 cycles, whereas Blind ML-PF2 algorithm results in a restored image shown in Fig. 22i and a reshaped OTF shown in Fig. 22j. More dramatic is the comparison of mean square errors in this case. Figs. 22k and 22*l* compare the MSE in object estimation and in OTF estimation for the three algorithms. The improvements resulting from the modified algorithms clearly illustrate the compensation for the effects of noise provided by the two modifications suggested here.

Fig. 22a

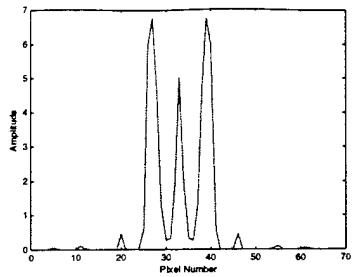


Fig. 22b

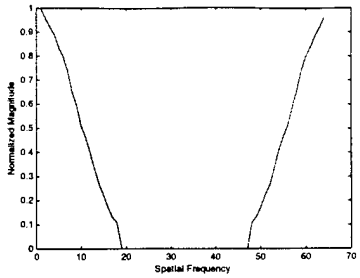


Fig. 22c

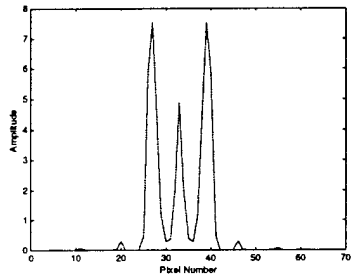


Fig. 22d

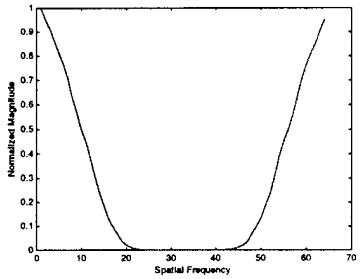


Fig. 22e

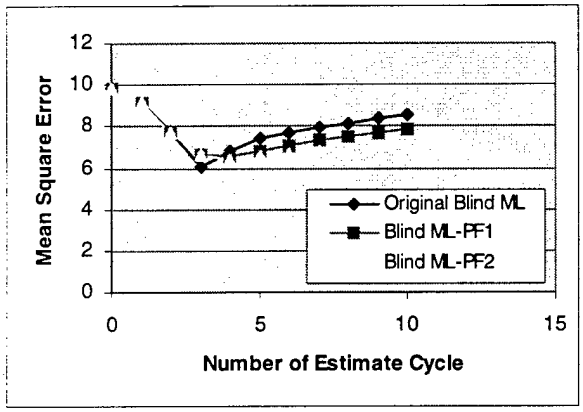


Fig. 22f

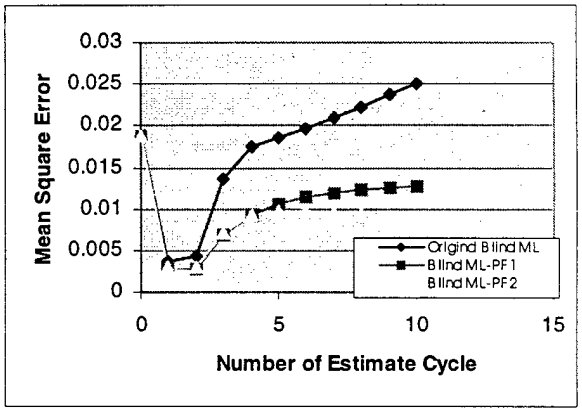


Fig. 22g

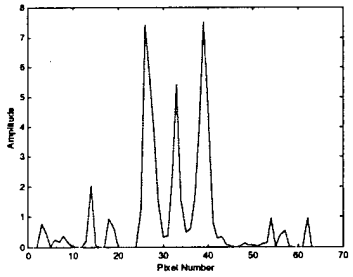


Fig. 22h

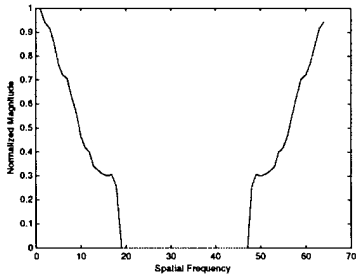


Fig. 22. Results of processing with Blind ML-PF1 and Blind ML-PF2 Algorithms

Fig. 22i

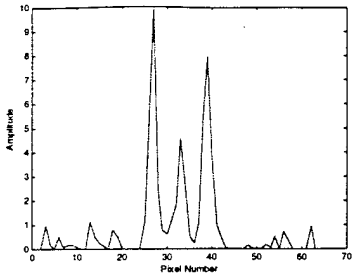


Fig. 22j

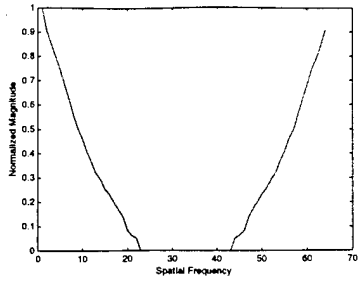


Fig. 22k

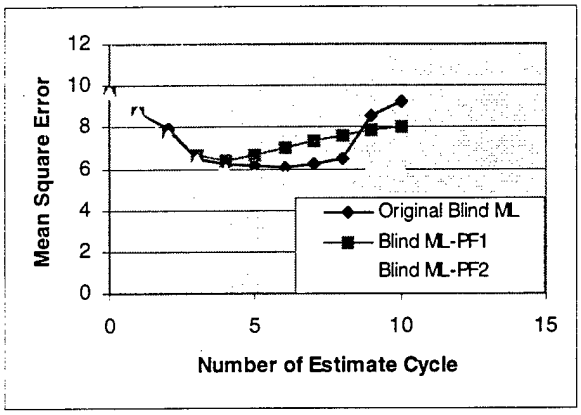


Fig. 22l

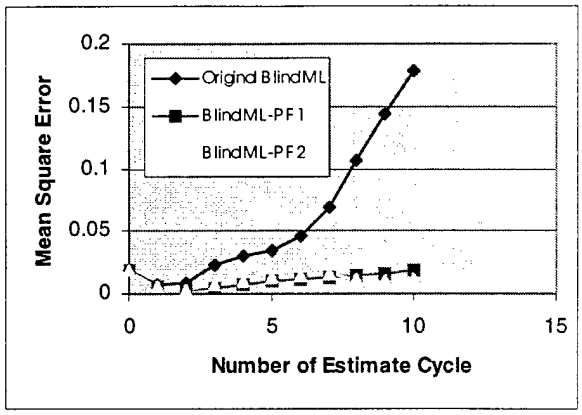


Fig. 22 (continued). Results of processing with Blind ML-PF1 and Blind ML-PF2 Algorithms

Experiment 2: The performance of the modified algorithms in processing noisy images was also tested by processing a 256x256 image from our database. Fig. 23a shows the original image used in this experiment. Fig. 23b shows the blurred image obtained by convolution with the PSF of a sensor with cutoff $\omega_c = 63$ and addition of Gaussian noise. The SNR in this image is 10.1084 dB. The reconstructed image after 6 cycles of processing with the original blind ML algorithm (with $m = 5$ and $n = 2$) is shown in Fig. 23c, which shows a marked improvement in the resolution. However, some artifacts due to the significant amount of noise present in the input image are clearly visible in the restoration (see the dark line under the left eye, for illustration). Fig. 23d shows the restored image after 6 cycles of processing with Blind ML-PF1 algorithm, which shows that the resolution enhancements are retained while some of the noise-induced artifacts are successfully eliminated. The frequency extrapolation resulting from this algorithm to achieve superresolution can also be demonstrated by comparing the frequency spectra of the input blurred image (shown in Fig. 23f) and of the restored image (shown in Fig. 23g) with the spectrum of the original image (shown in Fig. 23e). The

bandwidth extension is clearly noticeable in the four corners of the spectrum in Fig. 23g when compared to the spectrum of the input image indicating that the Blind ML-PF1 algorithm is indeed super-resolving.

Fig. 23a

Fig. 23b

Fig. 23c

Fig. 23d

Fig. 23e

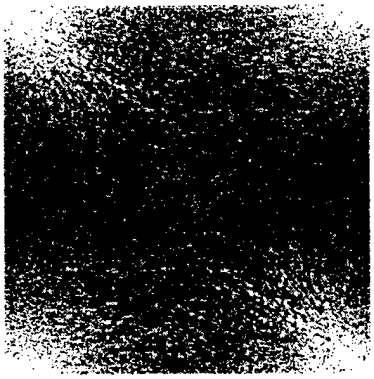


Fig. 23f

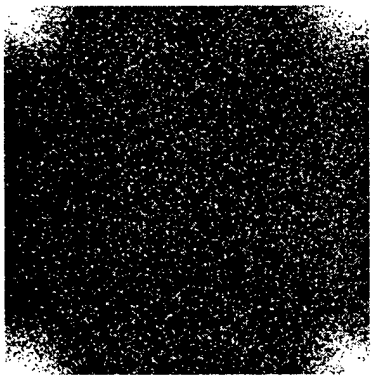


Fig. 23g

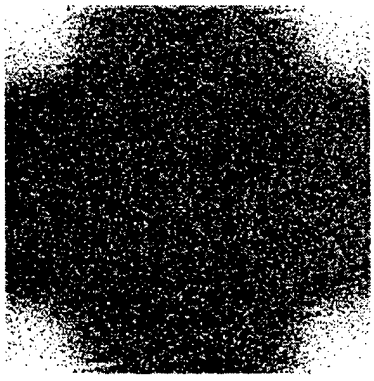


Fig 23. Results of super-resolution processing of two-dimensional image "Lenna"

4.3 Modified ML Algorithm With Background-Detail Separation

Since the primary goal of super-resolution processing is to enhance the resolution in the processed image through the creation of spatial frequencies of the object that are absent in the acquired image, an important measure of the performance of a super-resolution algorithm is the extent of spectral extrapolation it can provide. It has become quite well-known [11,13,28] that the amount of bandwidth extension beyond the cutoff frequency an algorithm can provide depends on certain characteristics of the image being processed, most notably the spatial extent of the object in the scene imaged. Experimental researchers have noted frustrating experiences of not being able to duplicate for their own images the bandwidth extension claims made by the developers of specific algorithms with carefully crafted examples of specifically tailored images. In particular, it has now become common knowledge that while very impressive bandwidth extensions are possible for images with spatially limited point sources against a uniform zero background (such as those arising in astronomical applications), spectral extrapolation to similar degrees cannot be obtained in general for images that contain objects which are not spatially limited, Unfortunately, a majority of images one encounters in military applications, particularly in target surveillance and tracking scenarios, fall into this latter category and typically contain several spatially distributed objects of interest (tanks, aircraft, military vehicles, etc.) sharing the scene with some background which may or nay not be uniform.

It has also been observed by some image processing researchers [31,32] that separation of the background from the detail component of the object imaged and subjecting these two components to different forms of processing can yield significant benefits. In particular, if the object being imaged permits a separation of these two components such that the background is uniform and extends over the whole extent of the scene, while the detail component is sparsely distributed (or nearly sparsely distributed), one can subtract the smooth background and attempt to restored only the detail component. Of particular significance to super-resolution processing from this strategy is the possibility of exploiting the spatially limited structures that may become available when one focuses on only the detail component. It is hence possible to develop optimized algorithms that provide an enhanced reconstruction of edges and other

features of targets of interest while suppressing the ringing artifacts these structures typically generate in the restoration process.

While the idea of background-detail separation outlined above seems conceptually simple, a number of intricate issues need to be addressed for a satisfactory implementation of a ML super-resolution algorithm utilizing this idea. First of all, due to the subtraction of the background, the detail component will not be nonnegative at all pixel locations and consequently appropriate measures (thresholding, for instance) need to be introduced to enforce non-negativity constraints. Secondly, since the detail component contains all the high frequency portions of the image (point sources, sharp edges, high frequency noise, etc.), growth of artifacts resulting from ML iterations of these need to be curtailed. A typical processing step that accomplishes this objective of suppressing the ringing (or the "halo effect" near sharp edges) is to introduce a soft thresholding operation [31,33] of the form

$$C\left(\hat{p}\right) = \alpha \hat{p} \frac{\left(\hat{p}\right)^2}{\beta + \left(\hat{p}\right)^2} \tag{51}$$

where α and β are appropriately chosen constants, after each iteration of obtaining the ML estimate $\hat{p}$. Finally, since the background component, which is processed independently, will be added to the estimate of the detail component, which is subjected to a nonlinear operation of the above form, in order to form the complete image estimate, appropriate constraints to ensure conservation of the total energy of the image [32] are to be introduced.

While several different algorithms can be constructed [33] by employing alternate procedures for enforcing the needed constraints as discussed above, we shall briefly outline a specific algorithm that has yielded superior performance especially in the context of processing PMMW images. The development of the algorithm proceeds with rewriting the model for the imaging process given by Equation (1) as

$$g(y) = \sum_{x \in X} h(y,x)\left[f^b(x) + f^d(x)\right] + noise \tag{52}$$

where $f^d(x)$ denotes the detail component and $f^b(x)$ denotes a smooth background component.

Since the background is smooth, the estimate of this component is relatively invariant to the effects of convolution with the sensor PSF and hence the estimate $f^b(j)$ for a discretized image can be obtained by severely blurring the image data, *i.e.*

$$\hat{f}^b(j) = h(j) \otimes g(j), \qquad j = 1,2,...,N \tag{53}$$

where N is the total number of pixels in the image. A ML estimate of the detail component is now obtained iteratively using the initial value $\hat{f}_0^d(j) = g(j) - \hat{f}^b(j)$ and progressively constructing the estimates $\hat{f}_k^d(j)$ by

$$\hat{f}_{n+1}^d(j) = \left(\hat{f}_n^d + \hat{f}_b\right)(j)\left[\left\{\frac{g(j)}{\left[\hat{f}_n^d + \hat{f}_b\right](j) \otimes h(j)}\right\} \otimes h(j)\right] - \hat{f}^b(j) \tag{54}$$

$$j = 1,2,...,N$$

For suppressing the artifacts, a soft threshold constraint is applied after the execution of each iteration by computing

$$C\left(\hat{f}_{n+1}^d\right) = \alpha \hat{f}_{n+1}^d \frac{\left(\hat{f}_{n+1}^d\right)^2}{\beta + \left(\hat{f}_{n+1}^d\right)^2} \tag{55}$$

with selected values of the constants α and β. The estimate of the complete image at this iteration is now computed as

$$\hat{f}_{n+1}(j) = C\left(\hat{f}_{n+1}^d(j)\right) + \hat{f}_b(j) \tag{56}$$

and constraints to enforce non-negativity and energy conservation are applied. The non-negativity constraint can be implemented simply from a thresholding operation,

$$\hat{f}_{n+1}(j) < 0, \quad => \quad \text{set } \hat{f}_{n+1}(j) = 0$$

$$\hat{f}_{n+1}(j) \geq 0, \quad => \quad \text{set } \hat{f}_{n+1}(j) = \hat{f}_{n+1}(j), \tag{57}$$

while the energy conservation constraint can be implemented by a scaling operation

$$\hat{f}_{n+1}(j) = \frac{\hat{f}_{n+1}(j)}{\sum_{j=1}^{N} \hat{f}_{n+1}(j)} \sum_{j=1}^{N} g(j) \tag{58}$$

The initial value for performing the next ML iteration is now obtained as

$$\hat{f}^{d}_{n+1}(j) = \hat{f}_{n+1}(j) - \hat{f}^{b}(j) \tag{59}$$

and the algorithm is repeated by executing the operations described by Equations (54)-(59).

While the ML estimation performed in Equation (54) is similar to that in the basic algorithm (given by Equation (24)), the additional steps described by Equations (55)-(59) result in an optimized version that produces further improvements in resolution together with suppression of ringing artifacts. We will refer to this algorithm as ML-BD algorithm in later discussion. These operations, however, are not computation intensive and hence the processor requirements for implementing the algorithm do not change significantly. For a progressive correction of the sensor PSF *h(j)* along with the estimation of the object, a blind version of this algorithm can be implemented by including an updating of the PSF at the end of each object estimation iteration as in Equation (47), resulting in a Blind ML-BD algorithm.

A few words on the selection of parameters α and β for implementing the soft threshold constraint given by Equation (55) seems to be in order, These parameters need to be selected with care to obtain optimized results that ensure achieving the twin objectives of resolution enhancement and artifact suppression (which, as noted earlier, are often contradictory). Values of these parameters can be tailored to the specific class of images being processed from conducting simple preliminary analysis steps. For instance, in the selection of appropriate values for these parameters for processing images containing man-made objects (such as the PMMW image "Humvee"), one may note that combinational geometry models are often appropriate for defining such objects and a useful description of each object can be obtained

from a combination of eight solid geometric primitives: rectangular parallelepiped, box, sphere, right circular cylinder, right elliptical cylinder, truncated right angle cone, ellipsoid of revolution, and right angle wedge [34]. By selecting the appropriate primitives and combining them, the shape and the location of any object can be described. For instance, to model the military vehicle in the PMMW image "Humvee", one may observe that the four primitives, rectangular parallelpiped, box, right angle wedge and right circular cylinder are sufficient. When mapping this 3-D model on a 2-D plane, the image can be viewed as a superposition of several one-dimensional blurred pulse signals. Thus an analysis for recovering the edges from these blurred pulses can be conducted and optimal values of α and β can be determined for implementing the ML-BD algorithm to ensure satisfactory edge recovery with minimal ringing artifacts. Following this analysis, the values of α and β were determined as $\alpha = 1$ and $\beta =$ -0.0005 which can further be used for processing the two-dimensional image.

Figs. 24a and 24b show the result of processing the PMMW image "Humvee" (acquired image shown in Fig. 4a) with ML-BD algorithm at the end of 3 iteration cycles. Comparing with the results earlier presented (Fig. 10), it may be noted that an improved resolution is achieved (which is evident from the details on the body of the vehicle and the sharper edges) while restoration artifacts are considerably eliminated (as evident from the restoration of the windshield and of the undercarriage of the vehicle).

Fig. 24a *Fig. 24b*

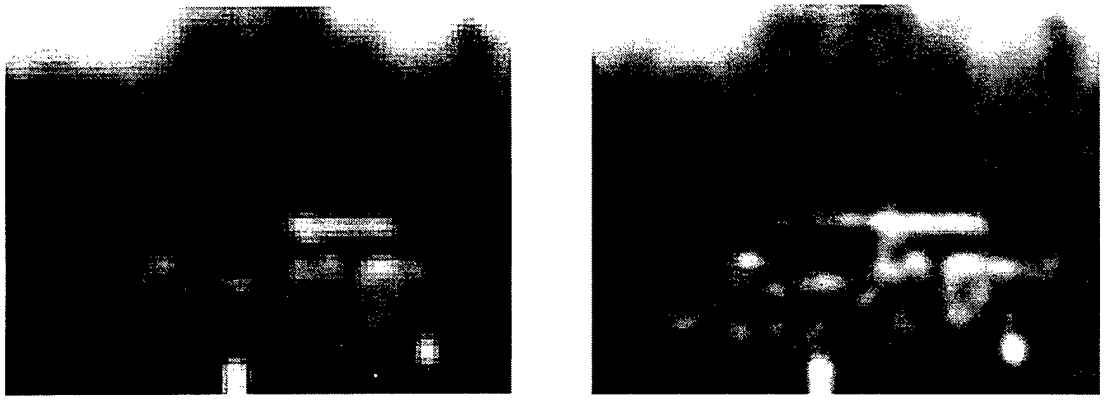


Fig. 24. Results of processing "Humvee" image by ML-BD algorithm

4.4 Modified Algorithm for Maximization of Posterior Density Function

The Maximum Likelihood (ML) restoration algorithm described by Equation (24) attempts to solve the inverse problem underlying the image restoration and super-resolution goals by employing a statistical modeling of the imaging process in order to optimize the "likelihood function". In certain cases when a mathematical model can be formulated for the prior distribution characterizing the object being estimated, which in the case of image restoration is the probability distribution of the intensity emanating from the object or the scene being imaged, it is possible to utilize this additional information to devise algorithms that provide improved performance as originally shown by Hunt [35]. Fundamental to this approach is a consideration of the posterior distribution as the quantity to be optimized (instead of the likelihood function which ML algorithms attempt to optimize). For an image formation process modeled by an input-output model of the form

$$g(x,y) = \Im(f(\tilde{x},\tilde{y})) \tag{60}$$

where $f(\tilde{x},\tilde{y})$ denotes the intensity function of the object being imaged defined over a region $(\tilde{x},\tilde{y})\varepsilon R_o$ and $g(x,y)$ denotes the intensity detected in the image over a region $(x,y)\varepsilon R_I$, Bayes rule relates these distributions in the form

$$p(f/g) = \frac{p(g/f)p(f)}{p(g)}. \tag{61}$$

In Equation (61), $p(f/g)$ denotes the posterior density function and $p(g/f)$ denotes the likelihood function, which are related through the prior distribution $p(f)$. It must be noted that with a statistical description of the imaging operation given by Equation (60), f denotes a set of random variables whose distribution $p(f)$ portrays the distribution followed by the photons right after they bounce off the object being imaged and before they travel to the imaging sensor. Thus, $p(f)$ summarizes the prior knowledge about the object and hence is

called the "prior density" (or simply the "prior"). Consequently, making use of this knowledge can steer the optimization process in the right direction and can in turn result in improved object estimates.

In MAP estimation, one attempts to find the estimate $\hat{f}$ that maximizes $p(f/g)$, *i.e.*

$$\hat{f} = \arg\max_{f} p(f/g). \tag{62}$$

Since $p(g)$ in Equation (61) is independent of f, this problem reduces to finding

$$\hat{f} = \arg\max_{f}\left[p(g/f)p(f)\right]. \tag{63}$$

Thus, depending on the statistical model used for $p(g/f)$ and $p(f)$, different estimates $\hat{f}$ can be obtained by solving this maximization problem.

While different distribution models could be used for modeling the prior $p(f)$, a Poisson distribution function results in a particularly simple algorithm. In this case $p(f)$ is modeled as

$$p\big(f(x,y)\big) = \prod_{x,y} \frac{\bar{f}(x,y)^{f(x,y)} e^{-\bar{f}(x,y)}}{f(x,y)!} \tag{64}$$

where $\bar{f}(x,y)$ denotes the mean photon emission rate at the location (x,y). As in the derivation of the ML restoration algorithm given earlier, the likelihood function is also modeled by a Poisson distribution

$$p(g/f) = \prod_{x',y'} \frac{(h \otimes f)^g e^{-h \otimes f}}{g!}$$

$$= \prod_{x',y'} \frac{\left(\sum_{x'',y''} h(x'-x'',y'-y'')f(x'',y'') \right)^{g(x',y')} e^{-\sum_{x'',y''} h(x'-x'',y'-y'')f(x'',y'')}}{g(x',y')!} \tag{65}$$

One may now proceed to solve the maximization problem posed in Equation (62) by taking the derivative of $p(f/g)$ and setting it equal to zero. As in the derivation of the ML algorithm, considerable simplification can be obtained by attempting to maximize the natural logarithm of $p(f/g)$. Thus, a MAP estimate $\hat{f}$ is sought by noting that

$$\ln p(f/g) = \ln p(g/f) + \ln p(f) - \ln p(g) \tag{66}$$

which results in the extremum condition

$$\frac{\partial \ln p(f/g)}{\partial f} = \frac{\partial \ln p(g/f)}{\partial f} + \frac{\partial \ln p(f)}{\partial f} = 0. \tag{67}$$

Employing the assumed distributions for $p(g/f)$ and $p(f)$ and using a simplification afforded by Stirling's approximation

$$\ln z! \approx z \ln z - z \tag{68}$$

one may evaluate the derivative on the Left Hand Side of Equation (67) as

$$\frac{\partial \ln p(f/g)}{\partial f} = \frac{\partial}{\partial f(x,y)}\left(\sum_{x',y'}\left(f(x',y')\ln \bar{f}(x',y') - \bar{f}(x',y') - f(x',y')\ln f(x',y') + f(x',y')\right)\right)$$
$$+ \frac{\partial}{\partial f(x,y)}\left(\sum_{x',y'}\left(g(x',y')\ln(h\otimes f)(x',y') - (h\otimes f)(x',y') - g(x',y')\ln g(x',y') + g(x',y')\right)\right)$$
$$= \ln \bar{f}(x,y) - \ln f(x,y) + \left(\frac{g(x,y)}{(h\otimes f)(x,y)} - 1\right) * h(x,y) \tag{69}$$

where the symbol $*$ denotes correlation.

Setting this quantity equal to zero, one now obtains the extremum condition in the simplified form

$$\ln \bar{f}(x,y) - \ln f(x,y) + \left(\frac{g(x,y)}{(h\otimes f)(x,y)} - 1\right) * h(x,y) = 0 . \tag{70}$$

Equation (70) is still complicated and hence one attempts to obtain an iterative solution in the form

$$\hat{f}^{n+1}(x,y) = F\left(\hat{f}^{n}(x,y)\right) \tag{71}$$

where $F(.)$ is the update function.

Taking the exponential on both sides of Equation (70) and rearranging the terms, one now obtains

$$f(x,y) = \bar{f}(x,y)\exp\left(\left(\frac{g(x,y)}{(h\otimes f)(x,y)} - 1\right) * h(x,y)\right) . \tag{72}$$

By making the assertion that $\bar{f}(x,y)$ is equal to the current estimate, the iterative updating rule for constructing MAP estimates is finally obtained in the form

$$\hat{f}^{n+1}(x,y) = \hat{f}^{n}(x,y)\exp\left[\left(\frac{g(x,y)}{(h\otimes f^{n})(x,y)} - 1\right) * h(x,y)\right]. \tag{73}$$

In order to start the iterative scheme, $\hat{f}^{0}(x,y) = g(x,y)$ is normally assumed.

It is interesting to note that the iterative algorithm specified by Equation (73) bears a close resemblance to the ML algorithm described by Equation (24). The principal difference is the presence of the exponential term in the MAP algorithm which provides an additional nonlinear processing term to permit generation of new frequencies thus facilitating improved spectral extrapolation in some cases. On the negative side, the presence of this term precludes making general assertions regarding the convergence of the iterations. Nevertheless, convergence has been noted in simulations, with even superior restoration results compared to the ML algorithm in several numerical experiments. As in the case of the ML algorithm, the speed of convergence is governed by several factors, most notably the accuracy of the knowledge of PSF and the SNR of the image. The MAP algorithm seems to have less robustness to PSF variations than the ML algorithm.

For a further comparison of the MAP algorithm with the more familiar ML algorithm, it is interesting to note that under certain conditions, the iterations specified by the updating rule given by Equation (73) approximate the updating executed in the implementation of the ML algorithm. Specifically, we observe that $\exp(\beta)$ can be approximated by the first two terms in its power series expansion, *viz.*, $1+\beta$, when the magnitude of β is small. Hence, when the numerical values of

$$\beta(x,y) = \frac{g(x,y)}{h\otimes f^{n}(x,y)} - 1 \tag{74}$$

attain small magnitudes, the Right Hand Side of Equation (73) can be replaced by

$$\hat{f}^{n}(x,y)\left[\frac{g(x,y)}{h\otimes \hat{f}^{n}(x,y)}\right] * h(x,y) \tag{75}$$

without introduction of much error. In this case, the updating required by the MAP algorithm will be identical to that required by the ML algorithm. One may also note that $\beta(x,y)$ in Equation (74) gives a measure of the residual error

$$\varepsilon(x,y) = g(x,y) - h \otimes \hat{f}^n(x,y) \tag{76}$$

at the end of the n^{th} iteration. Thus the performance of the MAP algorithm will approximately track the performance of the ML algorithm after the completion of a sufficient number of iterations when the residual error will hopefully have reduced to rather small values. However, this need not be the case in the initial steps of the algorithm implementation when the residual errors could have significantly large magnitudes.

It needs to be emphasized that successful restoration and super-resolution using MAP estimation approach can greatly depend on the question - which model to use for the prior? Geman and Geman [36] suggest the use of Markov Random Field (MRF) for reconstructing prior models which provide the ability to describe spatial correlation in the object intensity function. In a Markov model, each pixel is envisioned to loosely belong to some connected set of pixels (called "cliques") and a MRF prior can be formulated to specify any known information in the reconstruction by exploring the connectivity of pixels. Gibbs functions typically provide a powerful representation for MRF priors. The possibility of developing simple iterative updating rules such as that given by Equation (73) may however diminish with the use of such complex (albeit more accurate) prior functions in the maximization process, leaving one to settle for numerical optimization procedures. Furthermore, one may not be able to make any predictions on the convergence behavior of such implementations.

As mentioned earlier, restorations with the MAP algorithm in general yield very similar results comparable to the use of the ML algorithm. In specific cases one may expect to see some slightly improved performance however. An experiment that was conducted to evaluate the performance of the Map algorithm will be outlined now. Fig. 25a shows a PMMW image acquired by the ARL 95 GHz camera with a 3 *inch* lens/horn. This is the image ("pattern target") obtained at a range of about 200 *inches* of a 4 $ft \times 8$ *ft* specially constructed target

board with rectangular patches of radiation absorbing material. The restored image after 60 iterations of processing with the ML algorithm is shown in Fig. 25b. The enhancement of the edges and the sharpening of the features of the geometrical patterns are clearly visible in the processed image. The corresponding restoration obtained by the use of MAP algorithm is shown in Fig. 25c which depicts the result after 60 iterations of processing.

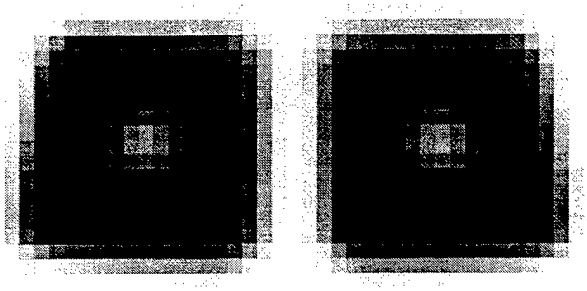


Fig. 25a. PMMW image "pattern target"

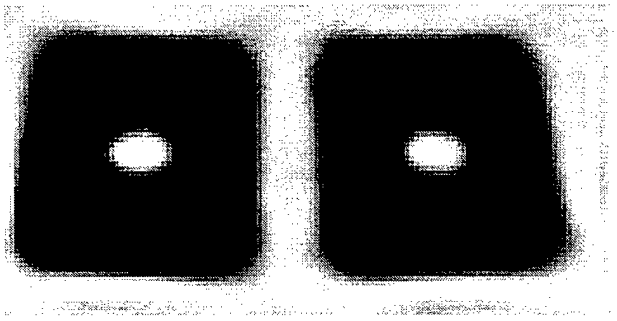


Fig. 25b. Restoration by ML algorithm

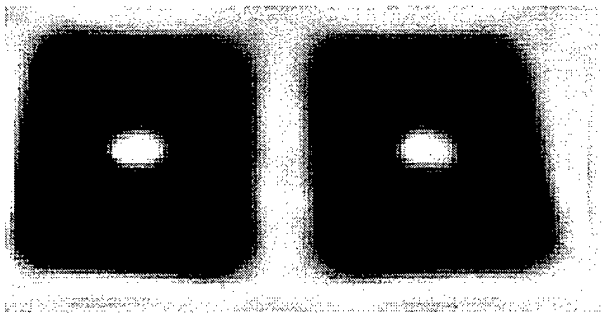


Fig. 25c. Restoration by MAP algorithm

Figs. 25a-c. Super-resolution of the "Test Pattern" image by MAP Algorithm

The restorations shown in Figs. 25b and 25c appear almost identical at first glance. Some quantitative calculations can be made to compare the results. It should be emphasized

that unlike in the case of the computation of resolution enhancement outlined earlier, a comparison with the ideal object is not possible in this case to calculate a parameter such as the virtual aperture diameter. Consequently, a more subjective evaluation may be needed in this case. One viable approach suggested in [41] is to extract a slice of data from the spatial frequency distribution plots for the processed and the unprocessed images and compare them. Fig.26 shows the variation of the normalized intensity (in dB) plotted as a function of the spatial frequency for a cut through the spectrum of the original acquired image in Fig. 25a (shown in darker line) and also the corresponding variation for the ML processed image in Fig. 25b (shown in fainter line). For specified intensity thresholds, one may identify the corresponding pixel numbers in the processed and the original data and compute the improvement by evaluating the ratio $pixel(processed) / pixel(unprocessed)$. These improvement factors computed at a few representative intensity levels are tabulated below for illustrative purposes.

Intensity level (dB)	**Pixel Unprocessed**	**Pixel Processed**	**Improvement Factor**
-40	8	18	2.25
-50	12	32	2.66
-60	20	32	1.60

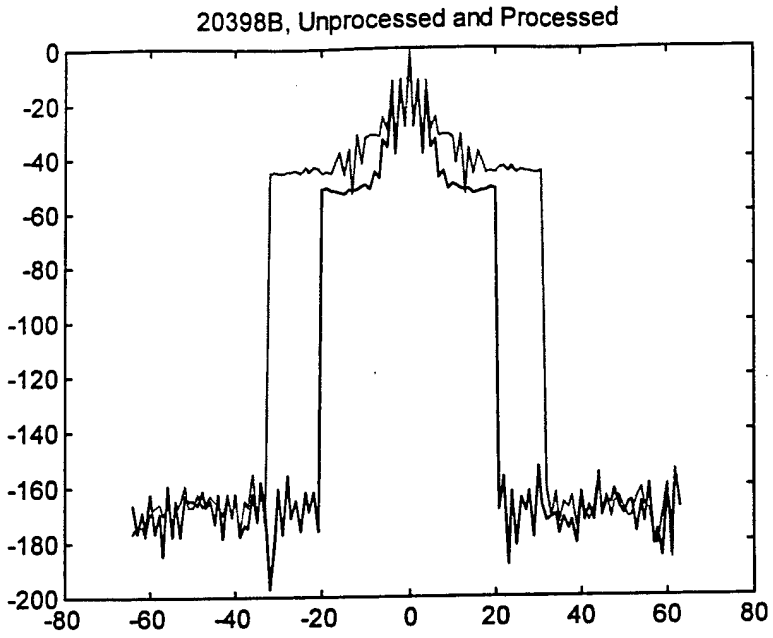


Fig. 26. Comparison of spatial frequency spectra of processed (ML) and unprocessed images.

Corresponding values of the Improvement Factor can be computed from the MAP processed image (in Fig. 25c) at the same intensity levels for a comparison. These values are shown tabulated below. The enhanced values for the MAP algorithm show that a somewhat improved restoration has been achieved in this case.

Intensity level (dB)	Pixel Unprocessed	Pixel Processed	Improvement Factor
-40	8	22	2.75
-50	12	35	2.92
-60	20	34	1.70

5. SELECTION OF SAMPLING RATES AND DESIGN OF OPTIMIZED SUPER-RESOLUTION ALGORITHMS WITH UPSAMPLING OPERATIONS

An aspect of particular importance in the digital implementation of restoration and super-resolution algorithms is the size of the sampling grid used for the representation of image data. Sampling at the Nyquist rate by a selection of the sampling pitch $p_x = p_y = \pi / \omega_c$, where ω_c is the cut-off frequency of the diffraction-limited imaging sensor, corresponds to an optimal spacing of detector elements or using a scan rate that provides the largest dwell time (thus enabling an increased SNR in the acquired image). However, selection of this grid size can lead to some problems when the Nyquist sampled image is subjected to super-resolution processing since the extrapolated frequencies will cause an overlap with the passband frequency components leading to aliasing of the spectrum. Consequently, one needs to work with oversampled images which provide a region of support for the creation of new frequencies such that these will not overlap with the existing frequency components. Sampling grid selection is hence a very important design consideration. In this section we shall outline our investigations on this issue which has led to the development of a progressive upsampling scheme for optimal implementation of super-resolution algorithms.

5.1 Image Representation on a Sample Grid

Image data typically used in processing operations will be in a sampled form. This could be a resultant of the hardware limitations in the data acquisition process which permit the imaging sensor to record the intensities only in a discrete form or it could be motivated by the desire for digital processing of data by using computer-based algorithms. The timing circuitry that controls the sampling of energy in the focal plane of the sensor is usually a very important part of the overall data acquisition hardware. For scan-type focal plane imaging sensors, the timing circuitry controls the acquisition of samples and also automatically adjusts the dwell time to match the scan velocity thus controlling the frame rate of the acquisition process.

The signal that represents the object or the scene being imaged, denoted $f(\tilde{x},\tilde{y})$ in Equation (1), is a continuous signal defined on a region $(\tilde{x},\tilde{y})\varepsilon R_o$ and the spectrum of this signal typically will have frequency components ranging from 0 to ∞. This signal when subjected to a low-pass filtering operation corresponding to diffraction limited imaging with a sensor with cutoff frequency ω_c, results in an image $g(x,y)$ which is also a continuous signal defined over a region $(x,y)\varepsilon R_I$. Due to convolution with the PSF of the sensor, the spectrum of $g(x,y)$ is band limited to the range $0 \le |\omega| \le \omega_c$, however.

As noted before, the recorded image data is in the sampled form. Let $g_s(x,y)$ denote the sampled version of the image and let p_x and p_y denote the sampling pitches in the x- and y-directions respectively. Ideally the image sampling process can be characterized as modulating the signal $g(x,y)$ with a set of impulse functions. Hence, denoting by M and N the number of samples collected in the x- and y-directions respectively, the sampled signal can be described by

$$g_s(x,y) = g\ (x,y)\sum_{m=1}^{M}\sum_{n=1}^{N}\delta\left(x-mp_x, y-np_y\right) \tag{77}$$

which can be rewritten as

$$g_s(x,y) = \sum_{m=1}^{M}\sum_{n=1}^{N} g\left(mp_x, np_y\right)\delta\left(x-mp_x, y-np_y\right) \tag{78}$$

where the set of values $g(mp_x, np_y)$, $m=1,2,3,\ldots M$ and $n=1,2,3,\ldots N$ constitute the image samples. For a concise representation, the sampling pitches p_x and p_y can be dropped and the sampled image can be described as

$$g(m,n) = g\left(mp_x, np_y\right). \tag{79}$$

It should be emphasized that $g(m,n)$ now denotes the sampled version of the original continuous signal $g(x,y)$.

To examine the effect of the sampling operation on the image spectrum, taking the Fourier transform of Equation (78) one obtains

$$G_s(j\Omega_x, j\Omega_y) = \frac{1}{p_x p_y} \sum_{k=-\infty}^{\infty} \sum_{l=-\infty}^{\infty} G_c(j\omega_x - kj\Omega_x^s, j\omega_y - lj\Omega_y^s) \tag{80}$$

where $\Omega_x^s = 2\pi / p_x$ and $\Omega_y^s = 2\pi / p_y$ denote the sampling frequencies (in *rad/m*) in the x- and y-directions. Equation (80) implies that the spectrum of the sampled image $g(m,n)$ is made up of superimposed and shifted copies of the spectrum of the nonsampled original image $g(x,y)$. If $g(x,y)$ is band-limited and the bandwidth extends only to a finite frequency value ω_h, and a sufficiently high rate of sampling is used to prevent an overlap of the shifted copies in $G_s(j\Omega_x, j\Omega_y)$, a reconstruction of the continuous image is possible from a low pass filtering of the sampled image. The low pass filter is applied to the sampled image such that only one copy of the spectrum of the original continuous image is preserved while the rest (which are repititions of this portion) are discarded, which results in the reconstructed signal

$$g_r(x,y) = \sum_{k=-\infty}^{\infty} \sum_{l=-\infty}^{\infty} g(k,l) h_r(x - kp_x, y - lp_y) \tag{81}$$

In Equation (81), $g_r(x,y)$ denotes the reconstructed continuous signal and $h_r(x,y)$ denotes the spatial response of an ideal low pass filter centered at $(0,0)$ in the frequency domain with a zero frequency gain equal to $p_x p_y$ and cutoff frequencies $-\pi/p_x$, π/p_x, $-\pi/p_y$, and π/p_y respectively. A reconstruction filter that achieves this objective is given by

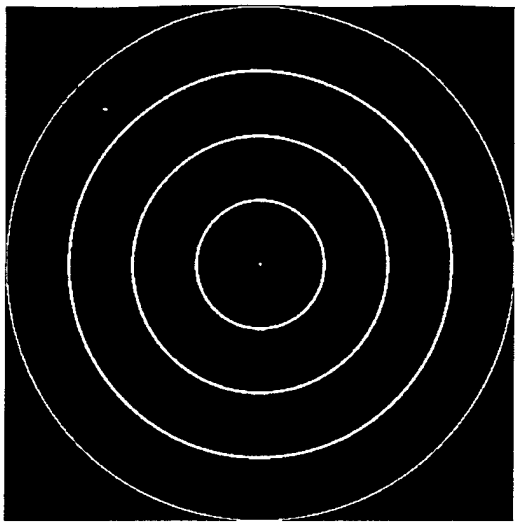
$$h_r(x,y) = \frac{\sin\dfrac{\pi x}{p_x}}{\dfrac{\pi x}{p_x}} \cdot \frac{\sin\dfrac{\pi y}{p_y}}{\dfrac{\pi y}{p_y}} . \qquad (82)$$

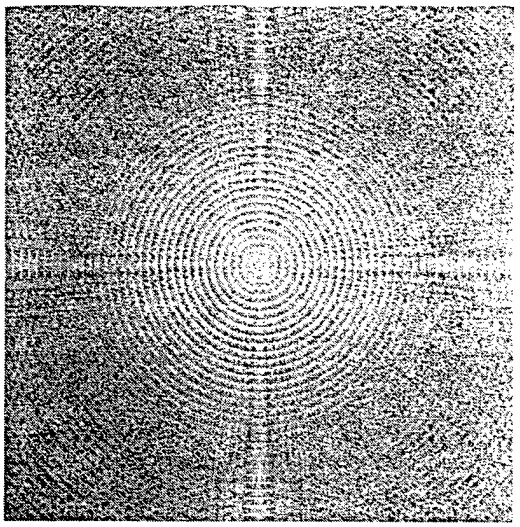
For a band-limited image with the highest spatial frequency ω_h, Nyquist theorem ensures that if the sampling frequencies are selected such that $\Omega_x^s \geq 2\omega_h$ and $\Omega_y^s \geq 2\omega_h$, there is no overlap of the spectral components in the sampled image. Images obtained from diffraction-limited sensing are necessarily band-limited to the sensor cutoff frequency ω_c. Hence no aliasing occurs if the sampling pitches are selected to satisfy the relations

$$p_x \leq \pi / \omega_c \quad \text{and} \quad p_y \leq \pi / \omega_c . \qquad (83)$$

Evidently, selection of the largest allowable value for p_x and p_y, *viz.*, $p_x = p_y = \pi / \omega_c$, results in the optimal number of detector elements to be used in the data acquisition process and also results in an image of a correspondingly small size, which is attractive from computational considerations. The sampling rate in this case is referred to as the Nyquist rate.

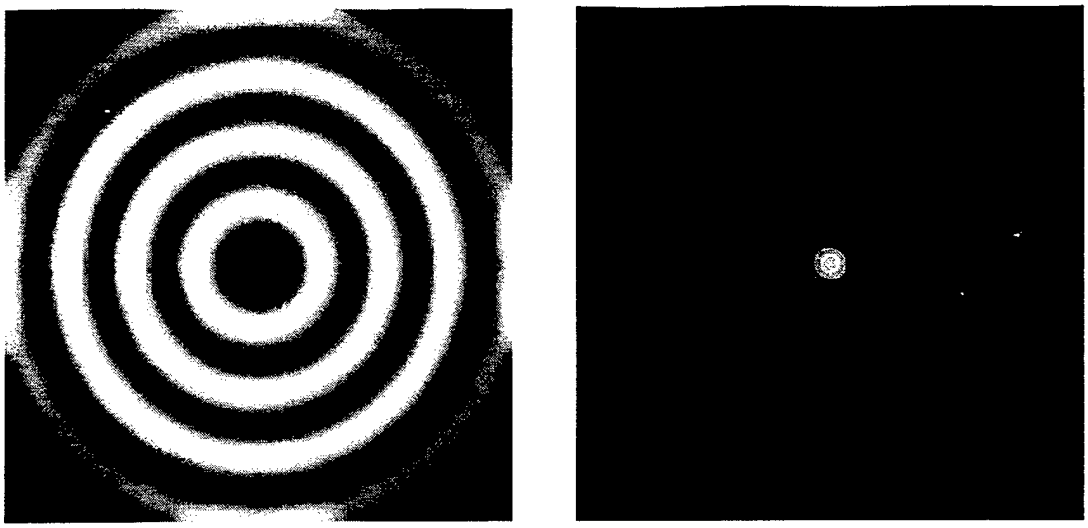
For illustrating the efficiency of representing an image on a selected grid of samples and the corresponding frequency-domain properties, let us again consider the "concentric disks" image described earlier in Section 3.2. For convenience, this is shown again in Fig. 27a. This is a simulated image of a series of concentric disks with the background (dark) at intensity value zero and the disks (bright) at intensity value one, and is represented on a sample grid of size 512×512. Since each of the transitions from a disk to the background and vice versa generates frequencies ranging from $-\infty$ to $+\infty$, the representation on a 512×512 grid of samples corresponds to undersampling. The aliasing of the spectral components can be seen in Fig. 27b, which displays the magnitude spectrum of this image. One may also note that while the aliasing is rather small and unnoticeable (since it is negligible compared to the numerical precision errors) at the center of the spectrum (low frequency range), some noticeable aliasing is present at the extremities (large frequencies).





Figs. 27a and 27b. Object represented on a 512 × 512 grid and its spectrum

Let us repeat the blurring operation earlier described in Chapter 3 on this object. Fig. 28a shows the blurred image resulting from the convolution of the image in Fig. 27a with an OTF of a sensor with a circular aperture of diameter 16 pixels. Since no frequency components beyond the cutoff frequency of 16 are present in the blurred image, a representation of this image on a grid of 512×512 samples constitutes oversampling (sampling rate exceeding the Nyquist rate). Hence no aliasing of spectral components will be present which is clearly seen in Fig. 28b. Fig. 29a shows a Nyquist sampled version of the blurred image which is a representation on a grid of 32×32 samples. Since the sampling rate is just adequate to prevent an overlap of spectral components, no aliasing results which can be seen in Fig. 29b. It is to be noted that the 512×512 image in Fig. 28a is a 8x oversampled version of the Nyquist sampled 64×64 image in Fig. 29a. Due to the preservation of integrity of spectral components, both images contain the same amount of information; indeed, one can be transformed into the other without any loss of information through "resampling" operations that will be discussed in a later section.



Figs. 28a and 28b. Blurred image and its spectrum

Figs. 29a and 29b. Blurred image represented on a 32× 32 grid and its spectrum

5.2 Need for Oversampling in Super-resolution Processing

While sampling at the Nyquist rate by a selection of the sampling pitch $p_x = p_y = \pi / \omega_c$ corresponds to an optimal spacing of detector elements or a scan rate that provides the largest dwell time (thus enabling an increased SNR in the acquired image), it will lead to some problems when the Nyquist sampled image is subjected to super-resolution processing. If the super-resolution algorithm employed successfully extrapolates the image bandwidth beyond the cutoff frequency ω_c, say up to an extended frequency limit $\omega_e (\omega_e > \omega_c)$, due to the periodic replication of spectral components the new frequencies created in the frequency range $\omega_c \le \omega \le \omega_e$ will overlap with the original frequency components present in $G_s\left(j\Omega_x, j\Omega_y\right)$. Not only will the new frequencies created become inaccurate but the original frequency components present in the passband of the image become corrupted due to the aliasing, which now extends over the entire frequency range $2\omega_c - \omega_e \le \omega \le \omega_e$. This in turn will produce aliasing artifacts in the restored image $\hat{f}(x,y)$ which is the output of super-resolution processing. Due to the corruption of the passband, the restored image may turn out to be poorer than the original image.

In order to prevent aliasing, the output $\hat{f}(x,y)$ must be represented on a grid of samples finer than the Nyquist sampled input image $g(m,n)$, which in turn requires a higher sampling rate to be used for the representation of the sampled image (thus increasing the image size). To determine the amount of oversampling required, one may anticipate the width of frequency extension that may reliably be expected from the super-resolution algorithm employed. Thus, if a 4x improvement in resolution is anticipated, one would expect to see the creation of object frequencies up to approximately $\omega_e = 4\omega_c$. Hence to ensure availability of support needed for this bandwidth extension, the image to be processed will need to be represented on a finer grid of samples corresponding to 4 times the Nyquist rate. This requirement in turn implies the selection of a smaller pitch $p_x = p_y = \pi / 4\omega_c$. The four-fold increase in the image size needs to be noted in this case.

Once the object estimate $\hat{f}(x,y)$ is generated as the output of super-resolution processing, it can be represented on the original coarser grid of samples corresponding to the Nyquist sampling rate by a downsampling operation. This step merely involves an aggregation of samples and requires simple averaging calculations. The need for downsampling or the desire to represent the constructed object estimate on a coarser grid of samples may arise simply from the efficiency it provides for storing and transmitting this data with reduced communication overhead. In several applications, such as in a multispectral missile seeker that employs a number of diverse sensors, reduction of communication overhead is of paramount importance for effective utilization of processed data (for illustration, for effective fusion of data coming from a number of sensor channels continuously into the fusion processor) [37].

5.3 Hardware and Software methods for Obtaining Oversampled data

Obtaining sampled image data at a rate higher than the Nyquist rate can be accomplished during data acquisition by modifying the hardware or as a post-acquisition signal processing step. In the latter case, additional software to perform signal resampling [4,38] may need to be included as part of the overall processing. PMMW sensor platforms currently being developed have attempted to include timing circuitry that provides the capability for acquiring oversampled imagery data [27,39]. While some recent demonstrations have discussed the

capability of collecting oversampled data (at a rate as high as 16 times the Nyquist rate), there are some significant tradeoffs this may entail, which in turn may render software methods considerably more attractive for accomplishing the desired goals. At the outset it must be noted that acquisition of oversampled data will typically require significant hardware modifications. One is forced to use either a larger focal plane array with more detector elements packed closely together, which incurs a higher cost, or attempt to include a precision scan mechanism which permits smaller scan steps through electronic scanning methods. In the latter case, it is important to note that scan speed is closely related to the integration time of the sensor and consequently has a major effect on the thermal sensitivity. Considering the fact that at the present state of development of PMMW imaging technology, the major problem is the slow response due to poor thermal sensitivity, increasing the scan speed may impose serious limitations on the quality of data (available SNR in the acquired image). Yet another method that is being investigated is the use of microscanning [40] which combines a number of slightly translated images into a single larger size image. Several techniques have been proposed to obtain microscanned PMMW images; all of these involve introduction of a subpixel dithering applied either to the PMMW antenna or to the focal plane array. Once again, the reduction in the frame rate and the integration time will pose serious limitations in this case as well. It must be noted in passing that microscanning is not necessarily used for oversampling always and even to obtain a sampled image at the Nyquist rate one may need to employ microscanning methods, As Bradley and Dennis [40] point out, focal plane arrays cannot be packed together closely enough to sample all of the spatial frequencies that they are able to detect and hence microscanning by dithering the images by half a detector pitch may be needed to obtain sampled data at the Nyquist rate.

In contrast to the above methods, signal processing methods provide a number of benefits for resampling of the image data. Resampling operations permit changing the number of pixels that define a sampled image in a new representation. If the total number of pixels is increased compared to the original grid, the operation is called "upsampling", while a downward revision of the total number of pixels leading to a coarser sample grid is called "downsampling". Upsampling typically involves interpolation of available data while

downsampling involves an averaging of data available. Introduction of upsampling operations can provide oversampled image data that can be readily used as input to super-resolution processing for avoiding aliasing artifacts. A considerable flexibility exists in upsampling a given sampled data set. For application to image data, a number of possible approaches exist and upsampling can be performed either on the space domain data directly or in the frequency domain using Fourier transform operations. The tradeoffs between processing complexity and the quality of the interpolated values need to be considered in selecting a specific approach for a given application.

Methods that employ space-domain operations for interpolation are generally less complex to implement. The simplest method is the zero-order hold, also called nearest neighbor assignment [38], which involves assigning to the interpolated element the value of the nearest known pixel. This is the fastest to implement of all the approaches. The next in the degree of complexity is the first-order hold, also called bilinear sampling [38], which uses the four closest known pixels that surround a new point to linearly interpolate its value. Evidently, the zero-order and first-order holds give piecewise and linear approximations respectively between the samples. Higher order approximations such as second-order and third-order holds give quadratic and cubic spline approximations. As the order of the hold approaches infinity, the interpolating function tends to a properly scaled Gaussian function [4]. It may also be noted that as the order of the hold increases, the interpolation error decreases (leading to better quality data), but the resolution loss will increase.

Frequency-domain methods for upsampling offer more attractive alternatives in terms of the quality of interpolated data. For a sampled image $g(m,n)$ of size $M \times N$, the Discrete Fourier Transform (DFT) coefficients that describe the image in the frequency domain are given by

$$G(k,l) = \sum_{m=0}^{M-1}\sum_{n=0}^{N-1} g(m,n) e^{-j\frac{2\pi mk}{M}} e^{-j\frac{2\pi nl}{N}} \quad , \quad k \in [0, M-1] \quad , \quad l \in [0, N-1]. \tag{84}$$

In order to upsample this image to produce a new image of size $\tilde{M} \times \tilde{N}$, where $\tilde{M} > M, \tilde{N} > N$, the image spectrum is zero padded to the desired size by defining the DFT coefficients in the form

$$\tilde{G}[k,l] = \begin{cases} \sum_{m=0}^{M-1}\sum_{n=0}^{N-1} g[m,n] e^{-j\frac{2\pi mk}{M}} e^{-j\frac{2\pi nl}{N}} & , \quad k \in [0, M-1] \quad and \quad l \in [0, N-1] \\ 0 & , \quad k \in [M, \tilde{M}-1] \quad or \quad l \in [N, \tilde{N}-1] \end{cases} \tag{85}$$

It is to be noted that the zero-padded spectrum is of dimension $\tilde{M} \times \tilde{N}$. The inverse DFT of $\tilde{G}(k,l)$ can now be calculated to yield the new image of size $\tilde{M} \times \tilde{N}$ by

$$\begin{aligned} \tilde{g}(m,n) &= \frac{1}{\tilde{M}\tilde{N}} \sum_{k=0}^{\tilde{M}-1} \sum_{l=0}^{\tilde{N}-1} \tilde{G}(k,l) e^{j\frac{2\pi \tilde{m}k}{\tilde{M}}} e^{j\frac{2\pi \tilde{n}l}{\tilde{N}}} \\ &= \frac{1}{\tilde{M}\tilde{N}} \sum_{k=0}^{M-1} \sum_{l=0}^{N-1} \tilde{G}(k,l) e^{j\frac{2\pi \tilde{m}(kM/\tilde{M})}{M}} e^{j\frac{2\pi \tilde{n}(lN/\tilde{N})}{N}} \end{aligned} \tag{86}$$

It can be shown that the zero padding in the frequency-domain as described above is equivalent to performing interpolation in the space-domain with a *sinc* function ($\sin x / x$). Furthermore, since all of the spectrum of the original image is retained in the new image, this interpolation does not introduce any approximation errors as in the case of space domain interpolation methods, excepting for the numerical errors related to the use of DFT and inverse DFT operations.

An illustration of the performance of the frequency-domain interpolation method discussed above can be given by an 16x upsampling of the 32×32 image shown in Fig. 29a. The upsampled 512×512 image is shown in Fig. 30a and the corresponding spectrum is shown in Fig. 30b. Comparing these to the image in Fig. 28a and its spectrum shown in Fig. 28b, one may note they are almost identical. The images are not identical in pixel values, however, due to the numerical errors introduced by DFT operation.

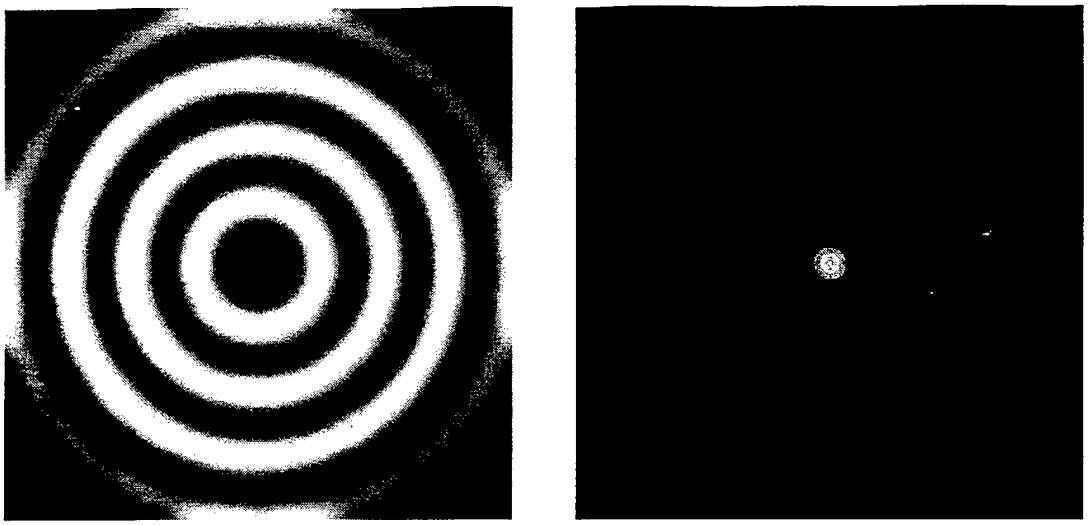


Fig. 30a and 30b. 16x upsampled version of image in Fig. 29a and its spectrum

Downsampling is the reverse of upsampling and produces an image of a reduced size. As in the case of upsampling, a given image can be downsampled by operations executed in the space-domain or in the frequency-domain. In the space-domain it involves averaging the pixel values, whereas in the frequency-domain one would reduce the spectrum to the desired reduced size by extracting the portion at the center of the spectrum of the original image. In particular, it must be noted that the frequency-domain upsampling does not eliminate any information present in the original image and hence it is always possible to recover the original image from its upsampled version. This however may not be true with the downsampling operation on a given image since this operation can throw away some information which may be impossible to recover from the downsampled version. It may also be noted that due to the use of DFT in going back and forth between the space- and frequency-domains, the upsampling and downsampling operations can increase or decrease the number of points used to describe the image only by powers of two.

How does the upsampling operation compare with hardware methods in obtaining oversampled data? Being a signal processing operation executed in software, upsampling is much simpler to implement and eliminates the costly modifications needed for hardware oversampling. In regard to the quality of data, while space-domain interpolation methods can introduce approximation errors (depending on the order of approximation), frequency-domain upsampling will not lead to any errors (other than the numerical errors in implementation of

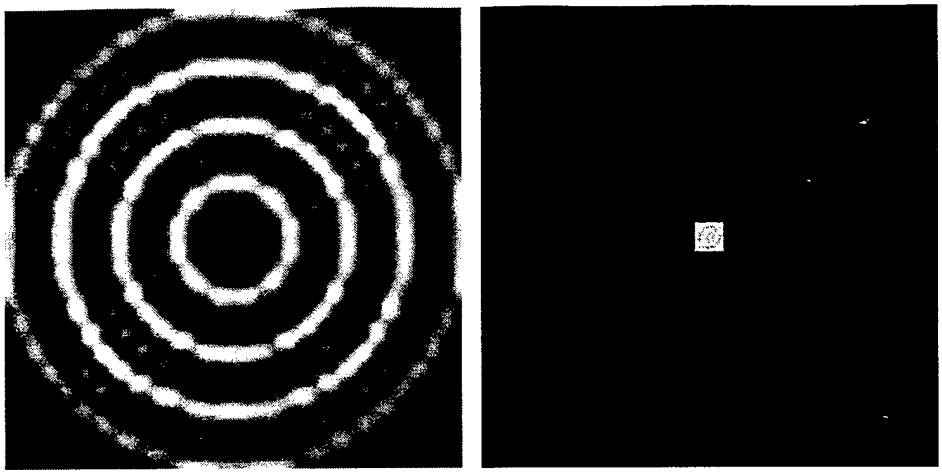
DFT operations, as noted earlier). In fact, it can be shown that for an image that is Nyquist sampled, under zero noise conditions the frequency-domain upsampling operation will provide an enlarged image that is comparable in accuracy to an image that is acquired by a sensor with a higher density of detector elements. This property together with the ease of implementation makes upsampling a more attractive alternative to hardware methods of oversampling which have the disadvantages of additional complexity and cost, reduced frame rate, and reduced integration time (leading to poor SNR in the acquired images).

5.4 Selection of Needed Degree of Oversampling

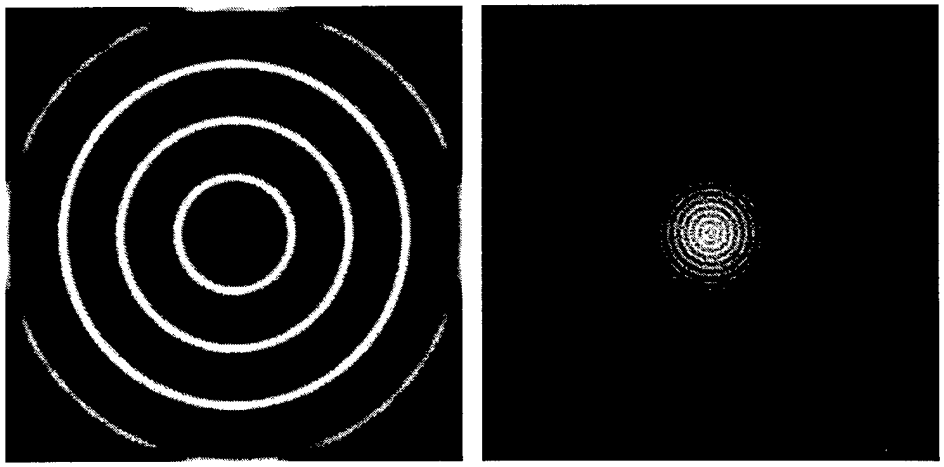
To obtain an understanding of what an oversampled version of data will do to the restoration and super-resolution process, an experiment was conducted by implementing the ML restoration algorithm given by Equation (24) on the blurred image shown in Fig. 29a. The Nyquist-sampled image of size 32×32 was processed by 50 iterations of the ML restoration algorithm. The resultant is also a 32×32 image. However, for a direct comparison, the processed image is also upsampled for representation on a 512×512 grid, which is shown in Fig. 31a. Comparing the images in Figs. 30a and 31a, any improvements are hardly noticeable. A further confirmation of this is also obtained by examining the spectrum of the processed image which is shown in Fig. 31b represented on a 512×512 grid. It is clear that not much super-resolution has resulted from the restoration processing, which is explainable from the lack of support available for the creation of new frequencies in this case. However, what seems to be more surprising is that not much of restoration seems to have taken place either due to the strong artifacts generated from processing.

In order to provide the needed support for the possible generation of new frequencies, upsampled versions of the blurred image were subjected to processing with the same number of iterations (specifically 50). To examine any differences in the achievable restoration, different degrees of oversampling (2x, 4x, 8x, and 16x resulting in images of sizes 64×64, 128×128, 256×256, and 512×512, respectively) were used. Fig. 32a shows the result of processing the 16x upsampled image which is displayed on the original grid size of 512×512 . The spectrum

of this image is shown in Fig. 32b. The expansion of the spectrum as well as the de-blurring effects are clearly visible in these figures.



Figs. 31a and 31b. Result of processing the Nyquist-sampled (64× 64) image and spectrum



Figs. 32a and 32b. Result of processing 16x upsampled (512× 512) image and its spectrum

In order to obtain a greater insight into the restoration performance observed in this experiment, the convergence behavior of the restoration error as well as the rate of increase in the likelihood function were examined. Since we are starting with a known object in this experiment (*i.e.,* image in Fig. 27a), one can easily compute the L_2-norm of the deviation between the object and the processed image as iterations progress. Furthermore at each successive iteration, the value of the likelihood function can be computed from noting that the logarithm of the likelihood function is given by

$$L(g/f) = \ln p(g/f) = \sum_{k=1}^{N}\left[g_k \ln\left(\sum_{j=1}^{N} h_{kj} f_j\right) - \sum_{j=1}^{N} h_{kj} f_j \right]. \qquad (87)$$

Figs. 33 and 34 display the convergence of the error and the increase in likelihood as iterations progress in the various cases. In Fig. 33, the top curve shows the decay of error in processing the Nyquist-sampled image, the middle curve when processing the 2x upsampled image, and the bottommost curve when processing the 4x, 8x and 16x upsampled images. It is of interest to note that while the processing of upsampled versions of the blurred image results in reduced restoration errors (which agrees with the explanation given above), no appreciable difference exists in the three cases of 4x, 8x and 16x oversampling although the images in the latter cases are considerably bigger than the former with corresponding increases in computational complexity. A further confirmation of this behavior is also evident from Fig. 34 where the lower curve shows the rate of likelihood increase for the Nyquist-sampled case while the upper curve shows the increase for the 2x, 4x, 8x and 16x oversampled cases. Clearly, there are no noticeable differences in the restoration of the 2x, 4x, 8x and 16x oversampled images.

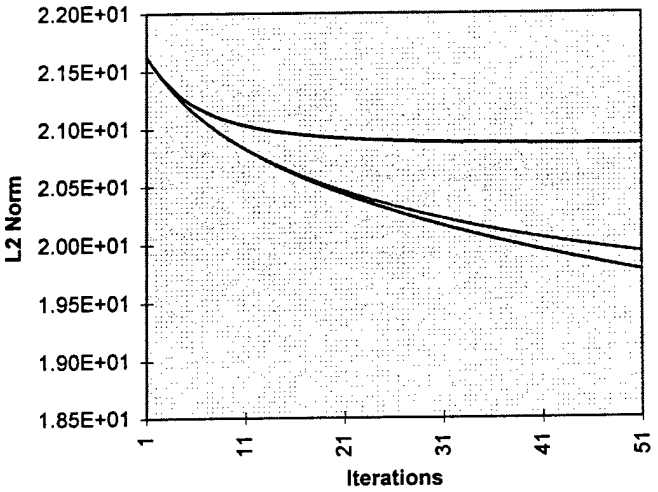


Fig. 33. Convergence of restoration error

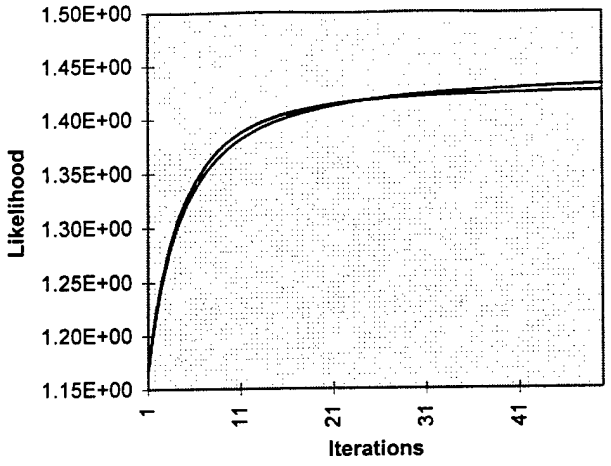


Fig. 34. Likelihood increase with progress of iterations

The last observation apparently has some important implications to several efforts currently being mounted for collection of oversampled data by implementing modifications to data acquisition circuitry, specially for sensing PMMW imagery data. Some recent studies [27,42] have proposed using microscanning techniques for acquiring PMMW data at 16x oversampled rate in order to achieve greater resolution gains through super-resolution processing. Our results evidently do not support such a proposition. For the simulation experiment discussed earlier, there is little need for oversampling beyond the 2x rate. Besides the need to process larger sized images, which increases the computational burden with little or no return in restoration performance, collection of oversampled data, either by increasing the density of detector elements in the sensor array or by microscanning techniques, will have detrimental effects on the image SNR due to the reduced integration times.

5.5 Design of a Progressive Upsampling Scheme for Iterative Restoration

From the discussion given in the earlier sections two important factors stand out. First, processing of an oversampled image is necessary in order to ensure any possible super-resolution from the iterative algorithm; however, an arbitrarily large degree of oversampling may not yield any benefits but may unnecessarily increase the computational burden. Second,

while the resolution improvements could be substantial during the first few iterations, the later iterations may not provide corresponding benefits and may in fact be not cost effective (considering the computational effort needed for implementing each iteration).

The two factors cited above suggest a more intelligent way of implementing the iterative algorithm by progressively increasing the rate of upsampling in order to provide the range of support required for frequency extension only as needed. Thus, one may commence the restoration of the Nyquist sampled blurred image and after a few iterations upsample by 2x to perform the next cycle of iterations. The rationale for working with the Nyquist sampled image at this step is that most of the improvement in the restored image comes from the passband restoration (*i.e.* for reversing the effects of convolution within the passband of the image) and hence the processing of the smallest sized image is useful at this step. The 2x upsampled image will have a size four times that of the original image but will also provide a frequency range $\omega_c \le \omega \le \tilde{\omega}_s / 2$ for the creation of new frequencies so that they will not overlap any frequencies within the image passband, where $\tilde{\omega}_s$ is the new sampling frequency corresponding to the 2x upsampled version of the image (specifically, $\tilde{\omega}_s = 2\omega_s$, where $\omega_s = 2\omega_c$ for the Nyquist sampled representation). Extrapolation of the object spatial frequencies, *i.e.* super-resolution, will begin at this step and can be continued until an iteration step where significant benefits from processing are no longer noticeable. The degree of upsampling can now be increased, say to 4x of the original image, and the iterations continued. A schematic that describes this scheme of implementing m iterations of the ML restoration algorithm followed by a 2x upsampling operation is shown in Fig. 35.

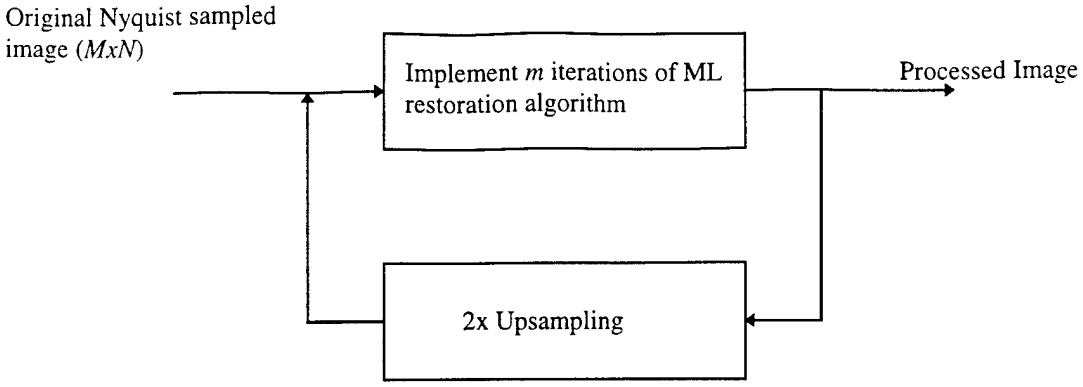
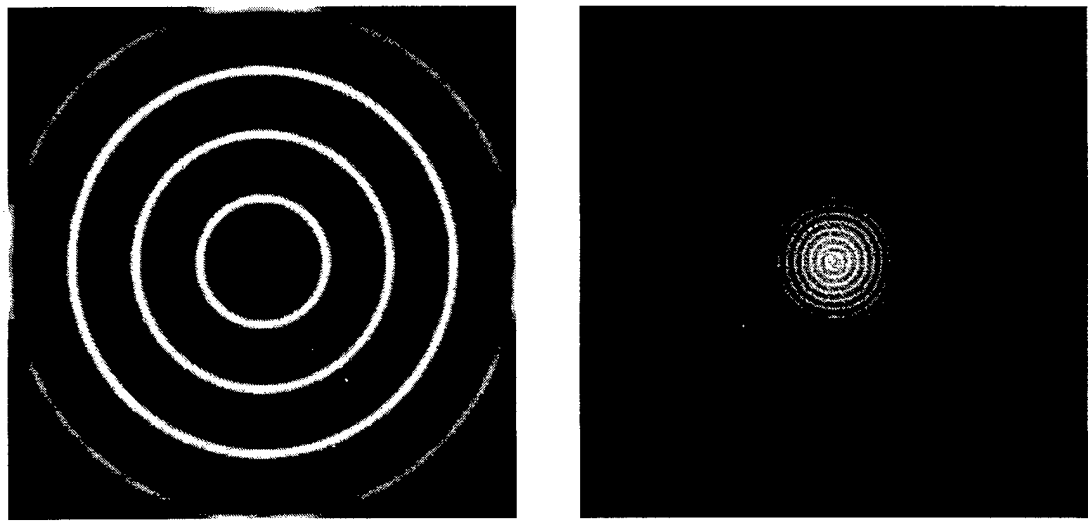


Fig. 35. The Progressive Upsampling Scheme for Implementation of ML Algorithm

For illustrating the performance of the progressive upsampling scheme described above, an experiment was conducted by performing ML restoration of the blurred "concentric disks" image that was considered earlier. Starting with a Nyquist-sampled representation of the blurred image of size 32×32, a first cycle of five ML iterations were executed and the result was upsampled to a 64×64 image. This image, which is a representation on a finer grid, was then subjected to the next cycle of five ML iterations. The result was upsampled to a 128×128 image and was then processed with the next cycle of five ML iterations after which it was upsampled again to a 256×256 image and subjected to the next cycle of five ML iterations. The result was again upsampled to yield a 512×512 image which was then processed iteratively for the next 30 iterations. The processed image at the end of this experiment is shown in Fig. 36a and its spectrum is shown in Fig. 36b.



Figs. 36a and 36b. Result of processing with progressive upsampling scheme and its spectrum.

A plot of the convergence of the L_2-norm of the deviation between the object and the processed image in this case as iterations progress is shown in Fig. 37. For comparison, a convergence plot of the L_2-norm in the case of processing the Nyquist-sampled version of the blurred image from the start of the iterations is also given in the same figure. It is of interest to note that the curve breaks downwards sharply at the point of upsampling due to the representation of the problem on a finer grid and the restart of a new iteration cycle, which exploits the property of faster convergence at the commencement of iterations as noted before. A second downward break, not as noticeable as the first one, also takes place at the next upsampling instant. The net effect of the progressive upsampling operation can hence be seen as a faster restoration.

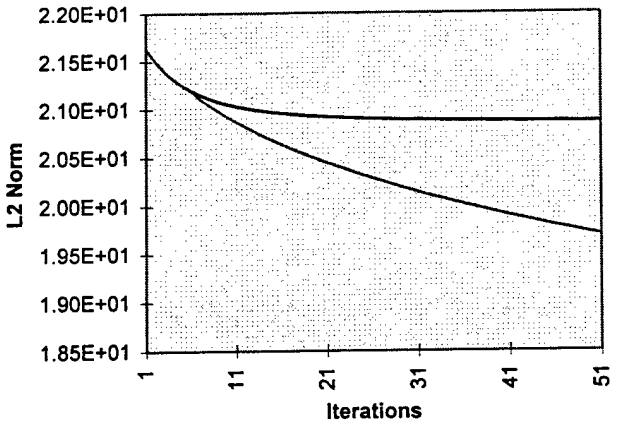


Fig. 37. Convergence of Restoration Error with Progressive Upsampling

A comparison of the spectrum in Fig. 36b with that of the blurred image in Fig. 28b gives a clear indication of the expanded frequency content and hence the super-resolution performance of the algorithm. Also, comparing the spectrum of the processed image with that in Fig. 32b, which displays the results of processing an 16x oversampled image (giving an image size of 512×512) for 50 iterations, it is clear that a comparable level of super-resolution performance is obtained with far less number of iterations. The increased computational

efficiency resulting from the need to process only images of smaller sizes initially and growing the size only as needed deserves to be noted.

The processing step at which to perform upsampling is largely a subjective decision. A guideline that can be used in the identification of an appropriate instant for upsampling is when the rate of decrease in the error (L_2-norm) has reduced to less significant values (*i.e.* the convergence has reached a relative plateau). While this evaluation based on the restoration error is possible when the object that resulted in the blurred image is known (as is the case in the present example), in practical image restoration tasks this is not available. In these cases however, one could examine the behavior of the likelihood function (given by Equation (87) with f replaced by the estimated signal $\hat{f}$) and base the decision on performing the upsampling at the points when the rate of increase in the likelihood function has reduced to a value less than what is desired based on the initial rate of increase achieved. An examination of the likelihood function can also be used to terminate the ML iterations at a processing step that yields the maximum value of the likelihood. A comparison of the increase in the likelihood function between the processing of the Nyquist sampled image for all 50 iterations without upsampling and with progressive upsampling is shown in Fig. 38.

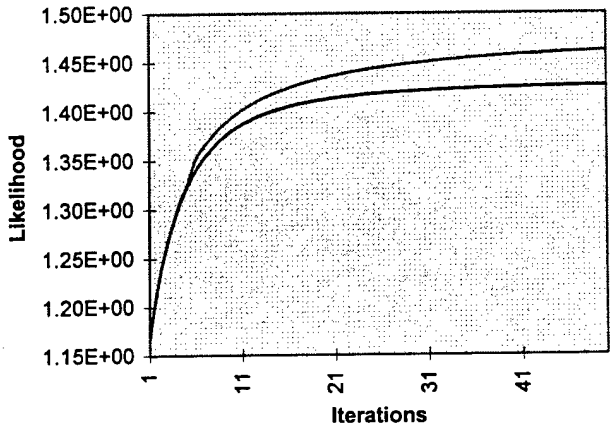


Fig. 38. Comparison of likelihood increase

In order to further validate the performance benefits resulting from the ML restoration algorithm aided with progressive upsampling, several restoration experiments on degraded images were conducted. In this section, we shall briefly outline the result from one of these experiments.

The super-resolution performance of the algorithm was tested by processing a 256×256 image from our database ("Lenna" image). Fig. 39a shows the original image used in this experiment. Fig. 39b shows the blurred image obtained by convolution with the PSF of a sensor with cutoff $\omega_c = 16$. The spectrum of the blurred image is shown in Fig.39c. For performing restoration, the blurred image is upsampled to a 512×512 image (with 16x oversampling) and subjected to 80 iterations of ML algorithm. The result of this processing is shown in Fig. 39d. The blurred image was also restored with progressive upsampling by commencing the iterative processing with an original 32×32 image (corresponding to a Nyquist sampling of the blurred image) and upsampling by 2x after executing every 5 iterations of ML algorithm until the final image size of 512×512 was obtained. The processing was terminated at the completion of the 40^{th} ML iteration from the start. The result of this processing is shown in Fig. 39e and the corresponding spectrum is shown in Fig. 39f. A comparison of the convergence performance in the two implementations is shown in Fig. 40, which displays a more attractive convergence behavior with the progressive upsampling scheme. In particular it is of interest to note that the error value with progressive upsampling at the 40^{th} iteration is smaller than the error at the 80^{th} iteration for processing the 16x oversampled image, which is obtained in addition to a significant saving in computational burden.

Fig.39a

Fig. 39b

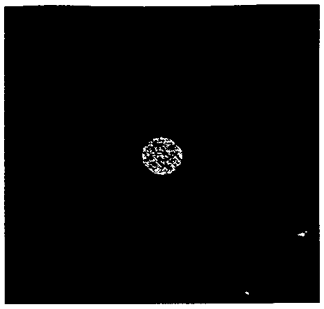


Fig. 39c

Fig. 39d

Fig.39e

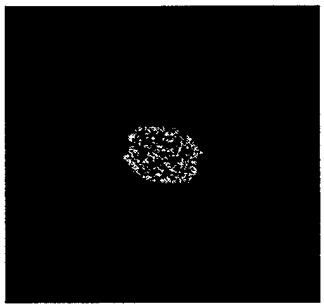
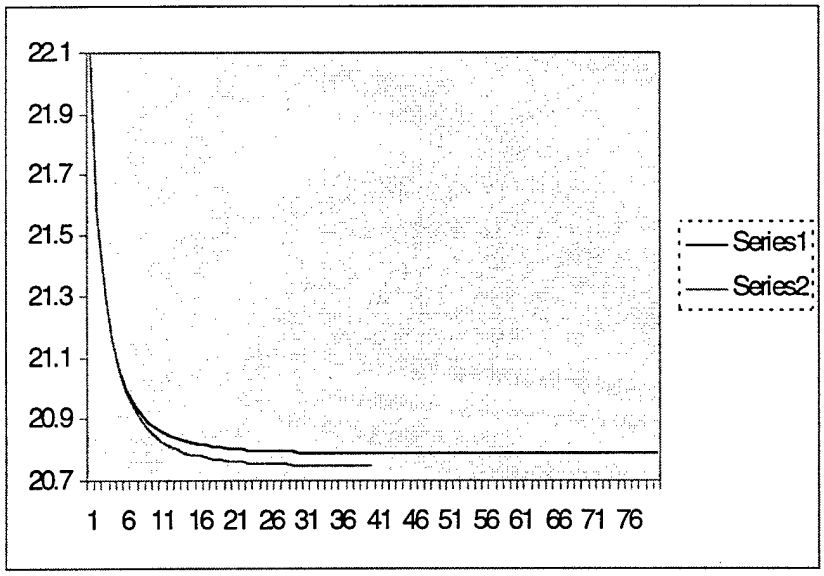


Fig. 39f

Fig. 39. Results of Processing "Lena" Image

Fig. 39a. Original Image; Fig.39b. Blurred Image; Fig. 39c. Spectrum of Blurred Image; Fig. 39d. Restored Image with no Upsampling; Fig. 39e. Restored with Progressive Upsampling; Fig. 39f. Spectrum of Restored Image by Progressive Upsampling;



Series 1: 80 iterations without upsampling
Series 2: 40 iterations with upsampling (every 5 iterations, 32x32 to 512x512)

Fig. 40. Comparison of Convergence Performance of the Two Implementations.

6. CONCLUSIONS

6.1 Summary of Results Obtained

This project was primarily aimed at the design and development of novel super-resolution algorithms for processing Passive Millimeter-wave (PMMW) imagery data. The principal objective of this effort was to evaluate and recommend specific techniques for super-resolution that will enable the U.S. Air Force to significantly enhance the usefulness of PMMW images it obtains for military intelligence purposes. Several investigations were conducted towards this objective results of which are described in this report.

During the current project, our investigations mainly focused on the design of iterative super-resolution algorithms that can be developed using a statistical modeling of the image formation process. The basic idea underlying these methods is to account for the statistical behavior of emitted radiation at the level of individual photon events by constructing appropriate object radiance distribution models (using knowledge of fluctuation statistics). Employing this mathematical framework enabled formulating the image restoration problem as mathematical optimization problems that yield Maximum Likelihood (ML) or Maximum A Posteriori (MAP) estimates of the restored object. Of particular interest to our work in this project, this framework also offered the feasibility of tailoring super-resolution algorithms whose spectrum extrapolation performance can be monitored and quantitatively measured. As established from previous work, two major factors that could limit successful implementations of image restoration and super-resolution algorithms in practice, particularly in missile seeker applications, are:

(1) lack of accurate knowledge of sensor point spread function (PSF) parameters, and

(2) noise-induced artifacts in the restoration process.

Our investigations hence addressed obtaining a characterization of these factors and focused on the design of restoration algorithms that possess a certain degree of robustness to inaccuracies in the knowledge of PSF parameters and also can suppress the noise-induced artifacts.

In order to facilitate performing restoration studies on PMMW imagery data, some efforts were also directed to building up a PMMW image database, and we now have a number of images acquired from several state-of-the-art PMMW radiometers developed by different hardware groups, specifically Air Force Research Labs at Eglin AFB, Army Research Lab at Adelphi, MD, and TRW, Inc. at Redondo Beach, CA. The studies that were conducted included a demonstration of resolution enhancements by processing various PMMW images in our database with these algorithms.

Several useful results have been obtained through our work on this project. Some of the more interesting ones will be briefly listed here:

- An analytical description of image restoration and super-resolution objectives from a spectrum reconstruction viewpoint has been developed which facilitates mathematically formulating the restoration and super-resolution problems in an optimization framework.
- Since satisfactory restoration and super-resolution of PMMW images can be achieved only through an intelligent tailoring of iterative processing algorithms, an analysis of some fundamental issues that need to be taken into account in synthesizing algorithms that afford simple digital implementation were conducted and several useful conclusions were developed. In particular, the important role played by the selection of a sampling grid size on the overall resolution achievable through digital processing of image data was examined in detail, which led to the formulation of a progressive upsampling procedure for optimized implementations of restoration and super-resolution algorithms.
- A database of PMMW imagery data collected using a number of different radiometer designs by different groups (AF Wright Lab Armament Directorate, TRW Inc., and Army Research Lab) has been developed. This database will be very valuable for conducting any future studies on the characterization and restoration of PMMW images.
- Using several PMMW images in this database, specific experiments are conducted to demonstrate the resolution enhancement and spectrum extrapolation resulting

from an iterative restoration algorithm developed using a Maximum Likelihood (ML) framework.

- A processor requirements analysis is conducted for evaluating the practical implementation of the iterative ML algorithm using commercial microprocessors in order to demonstrate that available chips such as TMS320C6x can support implementation of this algorithm while satisfying desired frame-rate requirements.
- Since image restoration involves essentially an inversion operation, the robustness of restoration algorithms to variations in the estimate of the sensor point spread function (PSF) is of particular importance. Several experiments are conducted to establish the robustness properties of the iterative ML super-resolution algorithm.
- For obtaining an iterative correction of the sensor PSF from a starting estimate, a blind deconvolution algorithm that jointly performs image restoration and PSF estimation is developed and the convergence properties of this algorithm are evaluated.
- Since the presence of significant amounts of noise in the measured data (poor SNR levels) is a source of generation of noise-induced restoration artifacts, a study of the dynamics of information transfer between the noise present and the image being restored during successive iterations of restoration processing is of particular importance. By conducting quantitative performance evaluations, the effects of noise on the processed image as well as on the accuracy of PSF estimation are examined.
- A modified ML algorithm that combines a post-filtering PSF adjustment step with the restoration processing is developed and its capabilities for suppressing noise-induced artifacts are established.
- Since the degree of spectrum extrapolation achievable from super-resolution processing depends on the spatial extent of objects in the scene, a procedure for separation of image background from the details is developed. This background-detail separation procedure facilitates subjecting only the detail portion of the image to super-resolution processing which in turn results in superior restoration of images.

- The problem of assessing the improvement in image quality subsequent to restoration processing is studied and various metrics in spatial and spectral domains are developed.
- Since the goal of super-resolution processing is to obtain a processed image that is equivalent to one obtained from a larger aperture (higher cost) sensor, a new metric that measures the "virtual aperture size" of the sensor is developed for quantifying the resolution improvement achieved from a restoration algorithm.

6.2 Some Directions for Further Research

Results obtained from this study can be improved by further investigations along several directions. One of the most promising directions seems to be the utilization of set theory-based estimation methods and integrating these with the statistical optimization approaches used in the present study. From our work on this project, as well as prior studies by others, it is now clearly understood that the quality and the extent of achievable super-resolution depends on the accuracy and the amount of *a priori* information that could be utilized in the solution process. The available information can be used to develop appropriate constraint sets that could steer the iterative estimation process in more efficient directions. In particular, the goals for further work should include the following:

- development of novel approaches based on convex constraint sets for resolution enhancement in the presence of noise and clutter, and for analysis of the limits to super-resolution;
- mathematical modeling of constraint sets and projection operators for specific *a priori* information about the object or scene being imaged in order to facilitate inclusion in the design of systematic restoration and super-resolution procedures based on the Projection-Onto-Convex-Sets (POCS) approach [43,44];
- investigation of methods for combining POCS-based approaches with the ML and MAP restoration methods discussed in this report for designing hybrid POCS-Bayesian super-resolution procedures.

REFERENCES

1. F. T. Ulaby, R. K. Moore and A. K. Fung, *Microwave Remote Sensing: Active and Passive, Vol. 1,* Artech House Inc., Massachusetts, 1981.

2. N. Skou, *Radiometer Systems; Design and Analysis,* Artech House Inc., Massachusetts, 1989.

3. J. Goodman, *Introduction to Fourier optics,* McGraw-Hill, 1996.

4. A. K. Jain, *Fundamentals of Digital Image Processing,* Prentice-Hall, 1989.

5. M. K. Sundareshan, " Advanced processing techniques for restoration and super-resolution of imagery in multispectral seeker environments", *Final Report to AFOSR Summer Faculty Research Program,* AF Wright laboratory Armament Directorate, Eglin AFB, FL, August 1995.

6. C.K. Rushforth and J.L. Harris, "Restoration, Resolution and Noise", *J. of Optical Society of America*, Vol. 58, pp. 539-545, 1968.

7. R.W. Gerchberg, "Super-resolution through error energy reduction", *Optica Acta*, Vol. 21, pp. 709-720, 1974.

8. A. Papoulis, "A new algorithm in spectral analysis and band-limited extrapolation", *IEEE Trans. on Circuits and Systems*, Vol. CAS-22, pp. 735-742, 1975.

9. K. Miller, "Least-squares method for ill-posed problems with a prescribed bound", *SIAM J. Math. Anal.*, Vol. 1, pp. 52-74, 1970.

10. D. Terzopoulis, "Regularization of the inverse visual problem involving discontinuities", *IEEE Trans. on Pattern Analysis and Machine Intelligence*, Vol. 1, pp. 413-424. 1986.

11. E.L. Kosarev, "Shannon's super-resolution limit for signal recovery", *Inverse Problems*, Vol. 6, pp. 55-76, 1990.

12. M. K. Sundareshan and P. Zegers, " Role of oversampled data in super-resolution processing and a progressive upsampling scheme for optimized implementations of iterative restoration algorithms", *Proc. of SPIE Conf. on Passive Millimeter-wave Imaging Technology, Aerosense'99*, Orlando, FL, April 1999.

13. P.J. Sementilli, B.R. Hunt and M.S. Nadar, "Analysis of the limit to super-resolution in incoherent imaging", *J. of Optical Society of America.*, Vol. 10. pp. 2265-2276, 1993.

14. M.I. Sezan and A.M. Tekalp, "Survey of recent developments in digital image restoration", *Optical Engineering*, Vol. 29, pp. 393-404, 1990.

15. J. Biemond, R.L. Lagendijk and R.M. Mersereau, "Iterative methods for image deblurring", *Proc. of IEEE*, Vol. 78, pp. 856-883, 1990.

16. M. K. Sundareshan, H.Y. Pang, S. Amphay, and B. Sundstrom, "Image restoration in multi-sensor missile seeker environments for design of intelligent integrated processing architectures," *Proc. of 1997 SPIE Conf. on Image Reconstruction and Restoration*, San Diego, California, August 1997.

17. A.M. Tekalp and M.I. Sezan, "Quantitative analysis of artifacts in linear space-invariant image restoration", *Multidimensional Systems and Signal Processing*, Vol. 1, pp. 143-177, 1990.

18. R.L. Langendijk, J. Biemond, and D.E. Bockee, "Regularized iterative image restoration with ringing reduction", *IEEE Trans. Acoustics, Speech and Signal Processing*, vol. ASSP-36, pp. 1874-1888, 1988.

19. N. C. Currie and C. E. Brown (Ed), *Principles and Applications of Millimeterwave Radar,* Artech House Inc., Massachusetts, 1987.

20. R. Appleby, D. G. Gleed, R. N. Anderton, and A. H. Lettington, 'Advances in passive millimeter wave imaging" *Proc. of SPIE,* Vol. 2211, pp 312-317, 1994.

21. W. H. Richardson, "Bayesian-based iterative method of image restoration," *J. Opt. Soc. Am.*, vol. 62, pp. 55-60, 1972.

22. L. B. Lucy, "An iterative technique for the rectification of observed distributions," *Astro. J.*, vol. 79, no. 6, pp. 745-759, June 1974.

23. L. Shepp and Y. Vardi, "Maximum likelihood reconstruction in positron emission tomography," *IEEE Trans. On medical Imaging*, vol. 1, pp. 113-122, 1982.

24. H. Y. Pang, M. K. Sundareshan, and S. Amphay, "Super-resolution of millimeter-wave images by iterative blind maximum likelihood restoration," *Proc. of 1997 SPIE Conf. On Passive Millimeter-Wave Imaging Technology*, Orlando, Florida, vol. 3064, pp. 227-238, April 1997.

25. H. Y. Pang, *Maximum Likelihood Restoration of Noisy Images*, M. S. Thesis, The University of Arizona, 1997.

26. H. Y. Pang, M. K. Sundareshan and S. Amphay, "Optimized maximum likelihood algorithms for super-resolution of passive millimeterwave imagery", *Proc. of 1998 SPIE Conf. on Passive Millimeterwave Imaging Technology*, Vol. 3378, pp 148-160, Orlando, FL, April 1998.

27. M. C. McKinley and D. D. Eden, " Oversampled passive millimeterwave images with application to image super-resolution", *Proc. of 1998 SPIE Conf. on Passive Millimeter-wave Imaging Technology*, Vol. 3378, pp 102-113, Orlando, FL, April 1998.

28. M.K. Sundareshan, "Performance of iterative and noniterative schemes for image restoration and super-resolution processing in multispectral seeker environments", *RDL Report SREP 96-0844,* Air Force Office of Scientific Research, Bolling AFB, February 1997.

29. D. G. Gleed and A. H. Lettington, "Application of super-resolution techniques to passive millimeter-wave images", *Proc. of 1991 SPIE Conf. on Applications of Digital Imaging Processing*, Vol. 1567, pp 65-72 Orlando, FL, April 1991.

30. DSP Specs, www.ti.com/sc/docs/dsps/products/c6x/benchmk.htm.

31. B. R. Frieden, "Image enhancement and restoration," in *Picture Processing and Digital Filtering*, T. S. Huang, ed., pp. 177-247, New York, Springer-Verlag, 1975.

32. B. R. Frieden, and D. C. Wells, "Restoring with maximum entropy. III Poisson sources and backgrounds," *Journal of the Optical Society of America,*, Vol. 68, No. 1, Jan. 1978.

33. M S. Nadar, P. J. Sementilli, and B. R. Hunt, "Estimation techniques of the background and detailed portion of an object in image susperresolution," *Proc. of 1994 SPIE Conf.*, Vol. 2241, pp. 204-215, 1994.

34. B. I. Hauss, H. Agravante and S. Chaiken, "Advanced radiometric millimeter-wave scene simulation ARMSS," *Proc. of 1997 SPIE Conf. On Passive Millimeter-wave Imaging Technology*, Vol. 3064, pp. 182-193, 1997.

35. B. R. Hunt, "Super-resolution of images: algorithms, principles, performance", *Int. Journal of Imaging Systems and Technologies,* Vol. 6, pp. 297-304, 1995.

36. S. Geman and D. Geman, "Stochastic Relaxation, Gibbs Distributions and the Bayesian Restoration of Images," *IEEE Trans. On Pattern Analysis and Machine Intelligence*, vol. 6, pp.721-741, 1984.

37. M. K. Sundareshan, "Fusion architectures for intelligent integrated processing of multisensor data in target surveillance and tracking applications", *Proc. of the 1998 Int. Conf. on Multisource-Multisensor Data Fusion (FUSION'98),* Las Vegas, NV, July 1998.

38. R. Schowengerdt, *Remote Sensing,* Academic Press, 1997.

39. A. R. Harvey, R. Appleby, P. M. Blanchard and A. H. Greenaway, "Beam-steering technologies for real time passive millimeter wave imaging", *Proc. of 1998 SPIE Conf. on Passive Millimeterwave Imaging Technology*, Vol. 3378, pp 63-72, Orlando, FL, April 1998.

40. D. J. Bradley and P. N. J. Dennis, "Sampling effects in HgCdTe focal plane arrays", *Proc. of 1985 SPIE Conf. on Infrared Technology and Applications,* Vol. 590, pp 53-60, 1985.

41. J. Silverstein, " Resolution and resolution improvement of passive millimeter-wave images", *Proc. of 1999 SPIE Conf. on Passive Millimeterwave Imaging Technology*, Vol. 3703, pp. 140-154, Orlando, FL, April 1999.

42. W. R. Reynolds, J. W. Hilgers and T. J. Schulz, " Super-resolved imaging sensors with field-of-view preservation", *Proc. of 1998 SPIE Conf. on Passive Millimeter-wave Imaging Technology*, Vol. 3378, pp. 134-147, Orlando, FL, April 1998.

43. D. C. Youla and H. Webb, " Image restoration by the method of convex projections: Part 1. Theory", *IEEE Transactions on Medical Imaging,* Vol. MI-1, No. 2, pp. 81-94, 1982.

44. M. I. Sezan and H. Stark, " Image restoration by the method of convex projections: Part 2. Application and numerical results", *IEEE Transactions on Medical Imaging,* Vol. MI-1, No. 2, pp. 95-101, 1982.

APPENDIX

In this Appendix, a complete listing of the computer programs and libraries developed in the IPDSL Laboratory for conducting various super-resolution experiments and investigations described in this report will be given. These programs are written in C language to facilitate easy use of these by others. A listing of the names of the various programs and a statement of what each program is designed to perform will be given first, which will be followed by the detailed code for each program.

A.1 List of Program Names, Libraries and Headers

- **Programs**

Program	Description
aperture.cpp	Creates an aperture
circles.cpp	Creates a test image with circles
convolve.cpp	Convolves two images
create.cpp	Creates an empty image
cutout.cpp	Cuts out a smaller image from another
dimensions.cpp	Shows the dimensions of an image
extreme.cpp	Shows the extreme values of an image
ima2pgm.cpp	Converts from IMA to PGM
insert.cpp	Inserts a small image into another
magspec.cpp	Calculates the magnitude of the spectrum
pattern.cpp	Creates a test image with rectangular patterns
pgm2ima.cpp	Converts from PGM to IMA
point.cpp	Creates a test image with a centered point
psf.cpp	Obtains a PSF from an aperture
resample.cpp	Resamples an image to any desired size
superresolution.cpp	Superresolves an image
threshold.cpp	Thresholds an image
totalflip.cpp	Applies the flip operator to an image

- **Libraries and Headers**

Library or Header	Description
dft.h	Header file for dft.cpp
dft.cpp	Routines needed to calculate the FFTs
pgm.h	Header file for pgm.cpp
pgm.cpp	Routines to convert to/from IMA to/from PGM
global.h	Header file with all global constants
image.h	Header file for image.cpp
image.cpp	Routines needed to manipulate an image
mcomplex.h	Header file for mcomplex.cpp
mcomplex.cpp	Routines needed to manipulate complex numbers

A.2 Description of Code

```
NAME
    aperture.cpp
USAGE
    aperture szApertureImaName inXDim inYDim szType inValue1 inValue2
INPUT PARAMETERS TYPE
    string              szApertureImaName
    integer             inXDim
    integer             inYDim
    string              szType
    integer             inValue1
    integer             inValue2
DESCRIPTION
    Creates an image of an ellipsoidal aperture. <szType> defines the type of aperture. <szType>
    has to be equal to ELLIPSOIDAL. The ellipse's horizontal axis is equal to <inValue1> pixels,
    its vertical axis equal to <inValue2> pixels. The aperture is stored in an image with
    <inXDim> by <inYDim> pixels. This image is stored in the file <szApertureImaName>.
PROGRAMMER
    Pablo Zegers
```

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "..\..\Tools\global\global.h"
#include "..\..\Tools\image\image.h"
#include "..\..\Tools\mcomplex\mcomplex.h"
#include "..\..\Tools\dft\dft.h"

int main(int ac, char *av[])
{
  char                                szBuffer[100];
  int                                 inStatus;
  IMAGE                              *pIMIma1;

  inStatus = OK;

  pIMIma1 = NULL;

  if (ac != 7)
  {
    printf("USAGE: aperture szApertureImaName inXDim inYDim szType inValue1 inValue2\n");
    inStatus = NK;
  }
  else
  {
    if (strcmp(av[4],"ELIPSOIDAL") == 0)
       {
      pIMIma1 = pIMIMAAskImaMem(atoi(av[2]),atoi(av[3]));

      inIMAElipsoidalAperture(pIMIma1,atoi(av[5]),atoi(av[6]));

      sprintf(szBuffer,"%s.ima",av[1]);
      inIMASaveIma(szBuffer,pIMIma1);
        }
  }

  if (pIMIma1 != NULL)
       inIMAFreeImaMem(pIMIma1);

  return(inStatus);
}
```

```
NAME
    circles.cpp
USAGE
    circles szMatrixName inSize
INPUT PARAMETERS
    string              szMatrixName
    integer             inSize
DESCRIPTION
    Creates a square image of <inSize> by <inSize> pixels with concentric circles and stores it
    in the file <szMatrixName>.
PROGRAMMER
    Pablo Zegers
```

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <math.h>
#include "..\..\Tools\global\global.h"
#include "..\..\Tools\image\image.h"
#include "..\..\Tools\mcomplex\mcomplex.h"
#include "..\..\Tools\dft\dft.h"

#define HEIGHT                  128
#define PERIOD                  64

int main(int ac, char *av[])
{
  char                                  szBuffer[100];
  int                                   inStatus,
                                                inSize,
                                                inCount1,
                                                inCount2;
  float                                 flRadius,
                                                flValue;
  mcomplex                              mcBuffer;
  IMAGE                               *pIMIma1;

  inStatus = OK;

  pIMIma1 = NULL;

  if (ac != 3)
  {
    printf("USAGE: circles szMatrixName inSize\n");
    inStatus = NK;
  }
  else
  {
    inSize = atoi(av[2]);

    pIMIma1 = pIMIMAAskImaMem(inSize,inSize);

        if (pIMIma1 == NULL)
        {
      printf("The image could not be created!\n");
      inStatus = NK;
        }
        else
        {
      for (inCount1=0;inCount1<inSize;inCount1++)
         {
        for (inCount2=0;inCount2<inSize;inCount2++)
              {
          flRadius = (float)sqrt((float)((inCount1-inSize/2)*(inCount1-inSize/2)+(inCount2-
inSize/2)*(inCount2-inSize/2)));

          flValue = (float)(128*cos((float)(2*PI*flRadius/((float)PERIOD))));

          if ((flRadius < inSize/2) && (flValue > 127))
            voSetCom(&mcBuffer,HEIGHT,0);
                else
            voSetCom(&mcBuffer,0,0);

          inIMASetImaElem(pIMIma1,inCount1,inCount2,mcBuffer);
              }
```

```
        }

        sprintf(szBuffer,"%s.ima",av[1]);
      inIMASaveIma(szBuffer,pIMIma1);

      inIMAFreeImaMem(pIMIma1);
        }
  }

  return(inStatus);
}
```

```
NAME
    convolve.cpp
USAGE
    convolve szResult szIma1 szIma2 szConTyp
INPUT PARAMETERS
    string              szResult
    string              szIma1
    string              szIma2
    string              szConTyp
DESCRIPTION
    Convolves the images stored in the files <szIma1> and <szIma2>. The string <szConTyp> defines
    the type of convolution: it can be LINEAR or CIRCULAR. The resulting image is stored in the
    file <szResult>.
PROGRAMMER
    Pablo Zegers
```

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "..\..\Tools\global\global.h"
#include "..\..\Tools\image\image.h"
#include "..\..\Tools\mcomplex\mcomplex.h"
#include "..\..\Tools\dft\dft.h"

int main(int ac, char *av[])
{
  char                                szBuffer[100];
  int                                 inStatus;
  IMAGE                              *pIMIma1,
                                      *pIMIma2,
                                      *pIMIma3;

  inStatus = OK;

  pIMIma1 = NULL;
  pIMIma2 = NULL;
  pIMIma3 = NULL;

  if (ac != 5)
  {
    printf("USAGE: convolve szResult szIma1 szIma2 szConTyp\n");
    inStatus = NK;
  }
  else
  {
    sprintf(szBuffer,"%s.ima",av[2]);
    pIMIma1 = pIMIMAReadIma(szBuffer);
        if (pIMIma1 == NULL)
        {
      printf("The image %s could not be read!\n",av[2]);
      inStatus = NK;
        }

    sprintf(szBuffer,"%s.ima",av[3]);
    pIMIma2 = pIMIMAReadIma(szBuffer);
        if (pIMIma2 == NULL)
        {
      printf("The image %s could not be read!\n",av[3]);
      inStatus = NK;
        }

        if (inStatus != NK)
        {
      pIMIma3 = pIMIMAAskImaMem(pIMIma2->inNumRows,pIMIma2->inNumCols);

          if (strcmp("LINEAR",av[4]) == 0)
       inIMAImaConvolution(pIMIma3,pIMIma1,pIMIma2,LINEAR);
          else if (strcmp("CIRCULAR",av[4]) == 0)
       inIMAImaConvolution(pIMIma3,pIMIma1,pIMIma2,CIRCULAR);
          else
       inStatus = NK;

          inIMAImaFunOp(pIMIma2,pIMIma3,IMRA);

      sprintf(szBuffer,"%s.ima",av[1]);
```

```
        inIMASaveIma(szBuffer,pIMIma2);
          }
    }

    if (pIMIma1 != NULL)
      inIMAFreeImaMem(pIMIma1);
    if (pIMIma2 != NULL)
      inIMAFreeImaMem(pIMIma2);
    if (pIMIma3 != NULL)
      inIMAFreeImaMem(pIMIma3);

    return(inStatus);
}
```

```
NAME
    create.cpp
USAGE
    create szImaName inHorSize inVerSize
INPUT PARAMETERS
    string              szImaName
    integer             inHorSize
    integer             inVerSize
DESCRIPTION
    Creates an empty image of <inHorSize> by <inVerSize> pixels. The image is stored in the file
    <szImaName>.
PROGRAMMER
    Pablo Zegers
```

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "..\..\Tools\global\global.h"
#include "..\..\Tools\image\image.h"
#include "..\..\Tools\mcomplex\mcomplex.h"
#include "..\..\Tools\dft\dft.h"

int main(int ac, char *av[])
{
  char                                szBuffer[100];
  int                                 inStatus;
  IMAGE                              *pIMIma1;

  inStatus = OK;

  pIMIma1 = NULL;

  if (ac != 4)
  {
    printf("USAGE: create szImaName inHorSize inVerSize\n");
    inStatus = NK;
  }
  else
  {
    pIMIma1 = pIMIMAAskImaMem(atoi(av[2]),atoi(av[3]));

    sprintf(szBuffer,"%s.ima",av[1]);
    inIMASaveIma(szBuffer,pIMIma1);
  }

  if (pIMIma1 != NULL)
    inIMAFreeImaMem(pIMIma1);

  return(inStatus);
}
```

```
NAME
    cutout.cpp
USAGE
    cutout szOutIma szInIma inX inY inDeltaX inDeltaY
INPUT PARAMETERS
    string              szOutIma
    string              szInIma
    integer             inX
    integer             inY
    integer             inDeltaX
    integer             inDeltaY
DESCRIPTION
    From the image stored in the file <szInIma>, cuts out another image whose upper left corner
    is positioned at (<inX>,<inY>). The size of this cut out image is <inDeltaX> by <inDeltaY>
    pixels.
PROGRAMMER
    Pablo Zegers
```

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "..\..\Tools\global\global.h"
#include "..\..\Tools\image\image.h"
#include "..\..\Tools\mcomplex\mcomplex.h"
#include "..\..\Tools\dft\dft.h"

#define UP                              0
#define DOWN                    1

int main(int ac, char *av[])
{
  char                                  szBuffer[100];
  int                                   inStatus;
  IMAGE                               *pIMIma1,
                                              *pIMIma2;

  inStatus = OK;

  pIMIma1 = NULL;
  pIMIma2 = NULL;

  if (ac != 7)
  {
    printf("USAGE: cutout szOutIma szInIma inX inY inDeltaX inDeltaY\n");
    inStatus = NK;
  }
  else
  {
    sprintf(szBuffer,"%s.ima",av[2]);
    pIMIma1 = pIMIMAReadIma(szBuffer);

        if (pIMIma1 == NULL)
        {
      printf("The image %s could not be read!\n",av[2]);
      inStatus = NK;
        }
        else
        {
          pIMIma2 = pIMIMAAskImaMem(atoi(av[5]),atoi(av[6]));

          inIMAExtractIma(pIMIma2,pIMIma1,atoi(av[3]),atoi(av[4]),atoi(av[5]),atoi(av[6]));

      sprintf(szBuffer,"%s.ima",av[1]);
      inIMASaveIma(szBuffer,pIMIma2);
        }
  }

  if (pIMIma1 != NULL)
    inIMAFreeImaMem(pIMIma1);
  if (pIMIma2 != NULL)
    inIMAFreeImaMem(pIMIma2);

  return(inStatus);
}
```

```
NAME
    dimensions.cpp
USAGE
    dimensions szIma
INPUT PARAMETERS
    string              szIma
DESCRIPTION
    Prints out the dimensions of the image stored in the file <szIma>.
PROGRAMMER
    Pablo Zegers
```

```
#include <stdlib.h>
#include <stdio.h>
#include "..\..\Tools\global\global.h"
#include "..\..\Tools\image\image.h"
#include "..\..\Tools\mcomplex\mcomplex.h"
#include "..\..\Tools\dft\dft.h"

int main(int ac, char *av[])
{
  char                                szBuffer[100];
  int                                 inStatus;
  IMAGE                              *pIMIma1;

  inStatus = OK;

  pIMIma1 = NULL;

  if (ac != 2)
  {
    printf("USAGE: dimensions szIma\n");
    inStatus = NK;
  }
  else
  {
    sprintf(szBuffer,"%s.ima",av[1]);
    pIMIma1 = pIMIMAReadIma(szBuffer);

      if (pIMIma1 == NULL)
      {
    printf("The image %s could not be read!\n",av[1]);
    inStatus = NK;
      }
      else
      {
    printf("%i rows by %i columns\n",pIMIma1->inNumRows,pIMIma1->inNumCols);
      }
  }

  if (pIMIma1 != NULL)
    inIMAFreeImaMem(pIMIma1);

  return(inStatus);
}
```

```
NAME
    extreme.cpp
USAGE
    extreme szIma szMinOrMax szExtTyp
INPUT PARAMETERS
    string              szIma
    string              szMinOrMax
    string              szExtTyp
DESCRIPTION
    Prints out the extreme value of the image stored in the file <szIma>. The parameter
    <szMinOrMax> defines if the extreme is a minimum or a maximum, and it has to be either MIN or
    MAX. The parameter <szExtTyp> has to be MAGNITUDE, REAL, or IMAGINARY according to the
    extreme that needs to be found.
PROGRAMMER
    Pablo Zegers
```

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "..\..\Tools\global\global.h"
#include "..\..\Tools\image\image.h"
#include "..\..\Tools\mcomplex\mcomplex.h"
#include "..\..\Tools\dft\dft.h"

int main(int ac, char *av[])
{
  char                                szBuffer[100];
  int                                 inStatus,
                                             inType;
  float                               flExtreme;
  IMAGE                            *pIMIma1;

  inStatus = OK;

  pIMIma1 = NULL;

  if (ac != 4)
  {
    printf("USAGE: extreme szIma szMinOrMax szExtTyp\n");
    inStatus = NK;
  }
  else
  {
    sprintf(szBuffer,"%s.ima",av[1]);
    pIMIma1 = pIMIMAReadIma(szBuffer);

       if (pIMIma1 == NULL)
       {
      printf("The image %s could not be read!\n",av[1]);
      inStatus = NK;
        }
        else
        {
      if (strcmp(av[3],"MAGNITUDE") == 0)
        inType = MAGNITUDE;
      else if (strcmp(av[3],"REAL") == 0)
        inType = REAL;
         else
        inType = IMAGINARY;

         if (strcmp(av[2],"MIN") == 0)
           inImaMin(&flExtreme,pIMIma1,inType);
         else
           inImaMax(&flExtreme,pIMIma1,inType);

         printf("The extreme is %.4f\n.",flExtreme);
       }
  }

  if (pIMIma1 != NULL)
    inIMAFreeImaMem(pIMIma1);

  return(inStatus);
}
```

```
NAME
    ima2pgm.cpp
USAGE
    ima2pgm szDesImaName szOriImaName
INPUT PARAMETERS
    string              szDesImaName
    string              szOriImaName
DESCRIPTION
    Converts the PGM P5 image stored in the file <szOriImaName> to the IMA format. The result is
    stored in the file <szDesImaName>.
PROGRAMMER
    Pablo Zegers
```

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "..\..\Tools\global\global.h"
#include "..\..\Tools\filters\filters.h"
#include "..\..\Tools\Image\Image.h"
#include "..\..\Tools\mcomplex\mcomplex.h"
#include "..\..\Tools\dft\dft.h"

int main(int ac, char *av[])
{
  char                                szBuffer[100];
  int                                 inStatus;
  IMAGE                              *pIMIma1,
                                      *pIMIma2;

  inStatus = OK;

  pIMIma1 = NULL;
  pIMIma2 = NULL;

  if (ac != 3)
  {
    printf("USAGE: ima2pgm szDesImaName szOriImaName\n");
    inStatus = NK;
  }
  else
  {
    sprintf(szBuffer,"%s.ima",av[2]);
    pIMIma1 = pIMIMAReadIma(szBuffer);

      if (pIMIma1 == NULL)
      {
     printf("The image %s could not be read!\n",av[2]);
     inStatus = NK;
      }
      else
      {
     sprintf(szBuffer,"%s.pgm",av[1]);
     pIMIma2 = pIMIMAAskImaMem(pIMIma1->inNumRows,pIMIma1->inNumCols);
     inIMAStretch(pIMIma2,pIMIma1,0,255);
     inFILSavePGM(szBuffer,pIMIma2,255);
      }
  }

  if (pIMIma1 != NULL)
    inIMAFreeImaMem(pIMIma1);
  if (pIMIma2 != NULL)
    inIMAFreeImaMem(pIMIma2);

  return(inStatus);
}
```

```
NAME
    insert.cpp
USAGE
    insert szOutIma szInIma inX inY
INPUT PARAMETERS
    string              szOutIma
    string              szInIma
    integer             inX
    integer             inY
DESCRIPTION
    Inserts the image stored in the file <szInIma> in the image stored in the file <szOutIma>.
    The upper left corner of the image to be inserted is positioned at (<inX>,<inY>). If some
    portions of the image to be inserted don't fit in the target image, they are clipped out. The
    resulting image is stored in the file <szOutIma>.
PROGRAMMER
    Pablo Zegers
```

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "..\..\Tools\global\global.h"
#include "..\..\Tools\image\image.h"
#include "..\..\Tools\mcomplex\mcomplex.h"
#include "..\..\Tools\dft\dft.h"

#define UP                              0
#define DOWN                    1

int main(int ac, char *av[])
{
  char                                  szBuffer[100];
  int                                   inStatus;
  IMAGE                               *pIMIma1,
                                            *pIMIma2;

  inStatus = OK;

  pIMIma1 = NULL;
  pIMIma2 = NULL;

  if (ac != 5)
  {
    printf("USAGE: insert szOutIma szInIma inX inY\n");
    inStatus = NK;
  }
  else
  {
    sprintf(szBuffer,"%s.ima",av[1]);
    pIMIma1 = pIMIMAReadIma(szBuffer);

        if (pIMIma1 == NULL)
        {
      printf("The image %s could not be read!\n",av[2]);
      inStatus = NK;
        }
        else
        {
      sprintf(szBuffer,"%s.ima",av[2]);
      pIMIma2 = pIMIMAReadIma(szBuffer);

          if (pIMIma2 == NULL)
          {
        printf("The image %s could not be read!\n",av[3]);
        inStatus = NK;
          }
          else
          {
            inIMAInsertIma(pIMIma1,pIMIma2,atoi(av[3]),atoi(av[4]));

        sprintf(szBuffer,"%s.ima",av[1]);
        inIMASaveIma(szBuffer,pIMIma1);
          }
        }
  }
```

```
  if (pIMIma1 != NULL)
    inIMAFreeImaMem(pIMIma1);
  if (pIMIma2 != NULL)
    inIMAFreeImaMem(pIMIma2);

  return(inStatus);
}
```

```
NAME
    magspec.cpp
USAGE
    magspec szMagSpecImaName szImaName
INPUT PARAMETERS
    string              szMagSpecImaName
    string              szImaName
DESCRIPTION
    Calculates the magnitude of the spectrum domain of the image stored in the file <szImaName>.
    The resulting image is stored in the file < szMagSpecImaName>.
PROGRAMMER
    Pablo Zegers
```

```
#include <stdlib.h>
#include <stdio.h>
#include "..\..\Tools\global\global.h"
#include "..\..\Tools\image\image.h"
#include "..\..\Tools\mcomplex\mcomplex.h"
#include "..\..\Tools\dft\dft.h"

int main(int ac, char *av[])
{
  char                                  szBuffer[100];
  int                                   inStatus;
  mcomplex                              mcValue;
  IMAGE                               *pIMIma1,
                                          *pIMIma2;

  inStatus = OK;

  pIMIma1 = NULL;
  pIMIma2 = NULL;

  if (ac != 3)
  {
    printf("USAGE: magspec szMagSpecImaName szImaName\n");
    inStatus = NK;
  }
  else
  {
    sprintf(szBuffer,"%s.ima",av[2]);
    pIMIma1 = pIMIMAReadIma(szBuffer);

       if (pIMIma1 == NULL)
       {
      printf("The image %s could not be read!\n",av[2]);
      inStatus = NK;
       }
       else
       {
      pIMIma2 = pIMIMAAskImaMem(pIMIma1->inNumRows,pIMIma1->inNumCols);
      inIMADFTIma(pIMIma2,pIMIma1,DFT);
      inIMAImaFunOp(pIMIma1,pIMIma2,MAGNITUDE);
      voSetCom(&mcValue,1,0);
      inIMAImaScaOp(pIMIma2,pIMIma1,mcValue,ADD);
      inIMAImaFunOp(pIMIma1,pIMIma2,LOG);
      inIMAImaShift(pIMIma2,pIMIma1);

      sprintf(szBuffer,"%s.ima",av[1]);
      inIMASaveIma(szBuffer,pIMIma2);
       }
  }

  if (pIMIma1 != NULL)
       inIMAFreeImaMem(pIMIma1);
  if (pIMIma2 != NULL)
    inIMAFreeImaMem(pIMIma2);

  return(inStatus);
}
```

```
NAME
    pattern.cpp
USAGE
    pattern szMatrixName inSize
INPUT PARAMETERS
    string              szMatrixName
    integer             inSize
DESCRIPTION
    Creates a square image of <inSize> by <inSize> pixels with rectangular patterns and stores it
    in the file <szMatrixName>.
PROGRAMMER
    Pablo Zegers
```

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <math.h>
#include "..\..\Tools\global\global.h"
#include "..\..\Tools\image\image.h"
#include "..\..\Tools\mcomplex\mcomplex.h"
#include "..\..\Tools\dft\dft.h"

#define HEIGHT                  128
#define PIXSHI                  28

int main(int ac, char *av[])
{
  char                                  szBuffer[100];
  int                                   inStatus,
                                                inSize,
                                                inCount1,
                                                inCount2,
                                                inWidth1,
                                                inWidth2;
  float                                 flValue;
  mcomplex                              mcBuffer;
  IMAGE                               *pIMIma1;

  inStatus = OK;

  pIMIma1 = NULL;

  if (ac != 3)
  {
    printf("USAGE: pattern szMatrixName inSize\n");
    inStatus = NK;
  }
  else
  {
    inSize = atoi(av[2]);

    pIMIma1 = pIMIMAAskImaMem(inSize,inSize);

        if (pIMIma1 == NULL)
        {
      printf("The image could not be created!\n");
      inStatus = NK;
        }
        else
        {
      inWidth1 = (int)floor(inSize/5);
      inWidth2 = (int)floor(inSize/7);

      for (inCount1=0;inCount1<inSize;inCount1++)
         {
        for (inCount2=0;inCount2<inSize;inCount2++)
              {
          if (inCount1 < inWidth1)
                {
            flValue = 0;
                }
                else if ((inCount1 >= inWidth1) && (inCount1 < 4*inWidth1))
                {
          if (inCount2 < inWidth2)
                    {
```

```
                flValue = 0;
                        }
                     else if ((inCount2 >= inWidth2) && (inCount2 < 6*inWidth2))
                        {
                if (((inCount1 >= 2*inWidth1) && (inCount1 < 3*inWidth1) && (inCount2 >=
2*inWidth2+PIXSHI) && (inCount2 < 3*inWidth2+PIXSHI)) ||
                   ((inCount1 >= 2*inWidth1) && (inCount1 < 3*inWidth1) && (inCount2 >=
4*inWidth2-PIXSHI) && (inCount2 < 5*inWidth2-PIXSHI)))
                        {
                  flValue = 0;
                        }
                        else
                        {
                  flValue = HEIGHT;
                        }
                     }
                     else
                        {
                flValue = 0;
                        }
                   }
                   else
                   {
              flValue = 0;
                   }


           voSetCom(&mcBuffer,flValue,0);

           inIMASetImaElem(pIMIma1,inCount1,inCount2,mcBuffer);
                }
              }

           sprintf(szBuffer,"%s.ima",av[1]);
        inIMASaveIma(szBuffer,pIMIma1);

        inIMAFreeImaMem(pIMIma1);
         }
    }

    return(inStatus);
}
```

```
NAME
    pgm2ima.cpp
USAGE
    pgm2ima szDesImaName szOriImaName
INPUT PARAMETERS
    string              szDesImaName
    string              szOriImaName
DESCRIPTION
    Converts the IMA image stored in the file <szOriImaName> to the PGM P5 format. The result is
    stored in the file <szDesImaName>.
PROGRAMMER
    Pablo Zegers
```

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "..\..\..\Tools\global\global.h"
#include "..\..\..\Tools\filters\pgm.h"
#include "..\..\..\Tools\image\image.h"
#include "..\..\..\Tools\mcomplex\mcomplex.h"
#include "..\..\..\Tools\dft\dft.h"

int main(int ac, char *av[])
{
  char                                szBuffer[100];
  int                                 inStatus;
  IMAGE                              *pIMIma1;

  inStatus = OK;

  pIMIma1 = NULL;

  if (ac != 3)
  {
    printf("USAGE: pgm2ima szDesImaName szOriImaName\n");
    inStatus = NK;
  }
  else
  {
    sprintf(szBuffer,"%s.pgm",av[2]);
    pIMIma1 = pIMFILReadPGM(szBuffer);

        if (pIMIma1 == NULL)
        {
      printf("The image %s could not be read!\n",av[2]);
      inStatus = NK;
        }
        else
        {
      sprintf(szBuffer,"%s.ima",av[1]);
      inIMASaveIma(szBuffer,pIMIma1);
        }
  }

  if (pIMIma1 != NULL)
    inIMAFreeImaMem(pIMIma1);

  return(inStatus);
}
```

```
NAME
    point.cpp
USAGE
    point szMatrixName inSize
INPUT PARAMETERS
    string              szMatrixName
    integer             inSize
DESCRIPTION
    Creates a square image of <inSize> by <inSize> pixels with a poit on its center and stores it
    in the file <szMatrixName>.
PROGRAMMER
    Pablo Zegers
```

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <math.h>
#include "..\..\Tools\global\global.h"
#include "..\..\Tools\image\image.h"
#include "..\..\Tools\mcomplex\mcomplex.h"
#include "..\..\Tools\dft\dft.h"

#define HEIGHT                  128

int main(int ac, char *av[])
{
  char                                  szBuffer[100];
  int                                   inStatus,
                                               inSize,
                                               inCount1,
                                               inCount2;
  mcomplex                              mcBuffer;
  IMAGE                               *pIMIma1;

  inStatus = OK;

  pIMIma1 = NULL;

  if (ac != 3)
  {
    printf("USAGE: point szMatrixName inSize\n");
    inStatus = NK;
  }
  else
  {
    inSize = atoi(av[2]);

    pIMIma1 = pIMIMAAskImaMem(inSize,inSize);

        if (pIMIma1 == NULL)
        {
      printf("The image could not be created!\n");
      inStatus = NK;
        }
        else
        {
      for (inCount1=0;inCount1<inSize;inCount1++)
          {
       for (inCount2=0;inCount2<inSize;inCount2++)
              {
        if ((inCount1 == (int)floor(inSize/2)) && (inCount2 == (int)floor(inSize/2)))
         voSetCom(&mcBuffer,HEIGHT,0);
              else
         voSetCom(&mcBuffer,0,0);

        inIMASetImaElem(pIMIma1,inCount1,inCount2,mcBuffer);
             }
          }

        sprintf(szBuffer,"%s.ima",av[1]);
      inIMASaveIma(szBuffer,pIMIma1);

      inIMAFreeImaMem(pIMIma1);
        }
  }
```

```
  return(inStatus);
}
```

```
NAME
    psf.cpp
USAGE
    psf szPointSpreadFunction szAperture inConType
INPUT PARAMETERS
    string              szPointSpreadFunction
    string              szAperture
    integer             inConType
DESCRIPTION
    Creates a point spread function using the aperture defined by the file <szAperture>. The type
    of convolution used is defined by <inConType>, which has to be either LINEAR or CIRCULAR. The
    resulting image is stored in the file <szPointSpreadFunction>.
PROGRAMMER
    Pablo Zegers
```

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "..\..\Tools\global\global.h"
#include "..\..\Tools\image\image.h"
#include "..\..\Tools\mcomplex\mcomplex.h"
#include "..\..\Tools\dft\dft.h"

int main(int ac, char *av[])
{
  char                                szBuffer[100];
  int                                 inStatus,
                                             inConType;
  IMAGE                            *pIMIma1,
                                          *pIMIma2;

  inStatus = OK;

  pIMIma1 = NULL;
  pIMIma2 = NULL;

  if (ac != 4)
  {
    printf("USAGE: psf szPointSpreadFunction szAperture inConType\n");
    inStatus = NK;
  }
  else
  {
    sprintf(szBuffer,"%s.ima",av[2]);
    pIMIma1 = pIMIMAReadIma(szBuffer);

       if (strcmp("LINEAR",av[3]) == 0)
      inConType = LINEAR;
    else if (strcmp("CIRCULAR",av[3]) == 0)
      inConType = CIRCULAR;
       else
      inStatus = NK;

       if ((inStatus != NK) && (pIMIma1 != NULL))
       {
      pIMIma2 = pIMIMAAskImaMem(pIMIma1->inNumRows,pIMIma1->inNumCols);

      inIMAPSFFromAperture(pIMIma2,pIMIma1,inConType);

      sprintf(szBuffer,"%s.ima",av[1]);
      inIMASaveIma(szBuffer,pIMIma2);
       }
  }

  if (pIMIma1 != NULL)
    inIMAFreeImaMem(pIMIma1);
  if (pIMIma2 != NULL)
    inIMAFreeImaMem(pIMIma2);

  return(inStatus);
}
```

```
NAME
    resample.cpp
USAGE
    resample szDesImaName inRows inCols szOriImaName
INPUT PARAMETERS
    string              szDesImaName
    integer             inRows
    integer             inCols
    string              szOriImaname
DESCRIPTION
    Resamples the image stored in the file <szOriImaName> to another image of size <inRows> by
    <inCols> pixels. The resulting image is stored in the file <szDesImaName>.
PROGRAMMER
    Pablo Zegers
```

```
#include <stdlib.h>
#include <stdio.h>
#include "..\..\Tools\global\global.h"
#include "..\..\Tools\image\image.h"
#include "..\..\Tools\mcomplex\mcomplex.h"
#include "..\..\Tools\dft\dft.h"

int main(int ac, char *av[])
{
  char                                szBuffer[100];
  int                                 inStatus;
  IMAGE                              *pIMIma1,
                                      *pIMIma2;

  inStatus = OK;

  pIMIma1 = NULL;
  pIMIma2 = NULL;

  if (ac != 5)
  {
    printf("USAGE: resample szDesImaName inRows inCols szOriImaName\n");
    inStatus = NK;
  }
  else
  {
    sprintf(szBuffer,"%s.ima",av[4]);
    pIMIma1 = pIMIMAReadIma(szBuffer);

      if (pIMIma1 == NULL)
      {
     printf("The image %s could not be read!\n",av[4]);
     inStatus = NK;
      }
      else
      {
     pIMIma2 = pIMIMAAskImaMem(atoi(av[2]),atoi(av[3]));
     inIMAResample(pIMIma2,pIMIma1);

     sprintf(szBuffer,"%s.ima",av[1]);
     inIMASaveIma(szBuffer,pIMIma2);
      }
  }

  if (pIMIma1 != NULL)
    inIMAFreeImaMem(pIMIma1);
  if (pIMIma2 != NULL)
    inIMAFreeImaMem(pIMIma2);

  return(inStatus);
}
```

```
NAME
    superresolution.cpp
USAGE
    superresolution szResultName szIdealName szBlurred szPriorName szPSF szMethod szConType
    inUpStep inUpLev inRepStep inNumIter
INPUT PARAMETERS
    string          szResultName
    string          szIdealName
    string          szBlurred
    string          szPriorName
    string          szPSF
    string          szMethod
    string          szConType
    integer         inUpStep
    integer         inUpLev
    integer         inRepStep
    integer         inNumIter
DESCRIPTION
    Superresolves the image stored in the file <szBlurred>. It uses the images stored in the
    files <szPriorName> and <szPSF>, as the prior knowledge image and point spread function
    needed for the superresolution process. The parameter <szMethod> has to be ML or MAP
    according to the type of superresolution wanted. <szConType> defines the type of convolution
    and it has to be either LINEAR or CIRCULAR. <inNumIter> defines the number of iterations.
    <inUpStep> defines the number of iterations per each prograssive upsampling stage. If
    <inUpStep> is equal to zero, then there is not progressive upsampling. <inUpLev> defines the
    number of pregressive upsampling levels. <inRepStep> defines the number of iterations that
    have to pass before the blurred image is replaced by the superesolved image. If <inRepStep>
    is equal to zero no replacement occurs. If <szIdealName> is different from NULL, it should
    define the name of the file that contains the image to be used as reference in all the
    reference-dependent measures. The superresolved image is stored in the file <szResultName>.
PROGRAMMER
    Pablo Zegers
```

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "..\..\Tools\global\global.h"
#include "..\..\Tools\image\image.h"
#include "..\..\Tools\mcomplex\mcomplex.h"
#include "..\..\Tools\dft\dft.h"

int inMeasures(float  *pflLik, int inAlgTyp, IMAGE *pIMIdeal, IMAGE *pIMf, IMAGE *pIMg,
                        IMAGE *pIMh, IMAGE *pIMp, int inConType, int inFlag);

int main(int ac, char *av[])
{
  char                              szBuffer[100];
  int                               inStatus,
                                           inConType,
                                           inCount1,
                                           inUpStep,
                                           inUpLev,
                                           inRepStep;
  float                             flLik,
                                           flMaxLik;
  IMAGE                          *pIMIdeal,
                                      *pIMBaseg,
                                      *pIMg,
                                      *pIMp,
                                      *pIMBaseh,
                                      *pIMh,
                                      *pIMf,
                                      *pIMIma1,
                                      *pIMIma2,
                                      *pIMBest;

  inStatus = OK;

  pIMIdeal = NULL;
  pIMBaseg = NULL;
  pIMg = NULL;
  pIMp = NULL;
  pIMBaseh = NULL;
  pIMh = NULL;
  pIMf = NULL;
```

```
  pIMIma1 = NULL;
  pIMIma2 = NULL;
  pIMBest = NULL;

  if (ac != 12)
  {
    printf("USAGE: superresolution szResultName szIdealName szBlurred szPriorName szPSF szMethod
szConType inUpStep inUpLev inRepStep inNumIter\n");
    inStatus = NK;
  }
  else
  {
       if (inStatus == OK)
       {
      if (strcmp("NULL",av[2]) != 0)
         {
        sprintf(szBuffer,"%s.ima",av[2]);
        pIMIdeal = pIMIMAReadIma(szBuffer);
        if (pIMIdeal == NULL)
              {
          printf("The image %s could not be read!\n",av[2]);
          inStatus = NK;
              }
         }

      sprintf(szBuffer,"%s.ima",av[3]);
      pIMBaseg = pIMIMAReadIma(szBuffer);
      if (pIMBaseg == NULL)
         {
        printf("The image %s could not be read!\n",av[3]);
        inStatus = NK;
         }
      pIMg = pIMIMAAskImaMem(pIMBaseg->inNumRows,pIMBaseg->inNumCols);
      inIMACopyIma(pIMg,pIMBaseg);

      sprintf(szBuffer,"%s.ima",av[4]);
      pIMp = pIMIMAReadIma(szBuffer);
      if (pIMp == NULL)
         {
        printf("The prior %s could not be read!\n",av[4]);
        inStatus = NK;
         }

      sprintf(szBuffer,"%s.ima",av[5]);
      pIMBaseh = pIMIMAReadIma(szBuffer);
      if (pIMBaseh == NULL)
         {
        printf("The prior %s could not be read!\n",av[5]);
        inStatus = NK;
         }
      pIMh = pIMIMAAskImaMem(pIMBaseh->inNumRows,pIMBaseh->inNumCols);
      inIMACopyIma(pIMh,pIMBaseh);

         if (strcmp("ML",av[6]) == 0)
         {
           if (strcmp("LINEAR",av[7]) == 0)
         inConType = LINEAR;
       else if (strcmp("CIRCULAR",av[7]) == 0)
         inConType = CIRCULAR;
           else
         inStatus = NK;

           inMeasures(&flLik,ML,pIMIdeal,pIMg,pIMg,pIMh,pIMp,inConType,NK);
         }
      else if (strcmp("MAP",av[6]) == 0)
         {
           if (strcmp("LINEAR",av[7]) == 0)
         inConType = LINEAR;
       else if (strcmp("CIRCULAR",av[7]) == 0)
         inConType = CIRCULAR;
           else
         inStatus = NK;

           inMeasures(&flLik,MAP,pIMIdeal,pIMg,pIMg,pIMh,pIMp,inConType,NK);
         }
         else
```

```
  inStatus = NK;

pIMf = pIMIMAAskImaMem(pIMg->inNumRows,pIMg->inNumCols);
pIMBest = pIMIMAAskImaMem(pIMg->inNumRows,pIMg->inNumCols);

    inUpStep = atoi(av[8]);
    inUpLev = atoi(av[9]);
    inRepStep = atoi(av[10]);

    for (inCount1=0;inCount1<atoi(av[11]);inCount1++)
    {
         if ((atoi(av[8]) != 0) && (inUpStep == 0) && (inUpLev > 0))
         {
    pIMIma1 = pIMIMAAskImaMem(2*pIMg->inNumRows,2*pIMg->inNumCols);
    pIMIma2 = pIMIMAAskImaMem(2*pIMg->inNumRows,2*pIMg->inNumCols);

    inIMAResample(pIMIma1,pIMBaseg);
        inIMAImaFunOp(pIMIma2,pIMIma1,IMRA);
        inIMAFreeImaMem(pIMg);
    pIMg = pIMIMAAskImaMem(pIMIma1->inNumRows,pIMIma1->inNumCols);
    inIMACopyIma(pIMg,pIMIma2);

    inIMAResample(pIMIma1,pIMp);
        inIMAImaFunOp(pIMIma2,pIMIma1,IMRA);
        inIMAFreeImaMem(pIMp);
    pIMp = pIMIMAAskImaMem(pIMIma1->inNumRows,pIMIma1->inNumCols);
    inIMACopyIma(pIMp,pIMIma2);

    inIMAResample(pIMIma1,pIMBaseh);
        inIMAImaFunOp(pIMIma2,pIMIma1,IMRA);
        inIMAFreeImaMem(pIMh);
    pIMh = pIMIMAAskImaMem(pIMIma1->inNumRows,pIMIma1->inNumCols);
    inIMACopyIma(pIMh,pIMIma2);

    inIMAResample(pIMIma1,pIMf);
        inIMAImaFunOp(pIMIma2,pIMIma1,IMRA);
        inIMAFreeImaMem(pIMf);
    pIMf = pIMIMAAskImaMem(pIMIma1->inNumRows,pIMIma1->inNumCols);
    inIMACopyIma(pIMf,pIMIma2);

    inIMAResample(pIMIma1,pIMBest);
        inIMAImaFunOp(pIMIma2,pIMIma1,IMRA);
        inIMAFreeImaMem(pIMBest);
    pIMBest = pIMIMAAskImaMem(pIMIma1->inNumRows,pIMIma1->inNumCols);
    inIMACopyIma(pIMBest,pIMIma2);

           inIMAFreeImaMem(pIMIma1);
           inIMAFreeImaMem(pIMIma2);

    inUpStep = atoi(av[8]);
           inUpLev -= 1;
         }

         if ((atoi(av[10]) != 0) && (inRepStep == 0))
         {
    inIMACopyIma(pIMg,pIMf);
           inRepStep = atoi(av[10]);
         }

         if (strcmp("ML",av[6]) == 0)
         {
        if (strcmp("LINEAR",av[7]) == 0)
      inConType = LINEAR;
    else if (strcmp("CIRCULAR",av[7]) == 0)
      inConType = CIRCULAR;
       else
      inStatus = NK;

           inIMAMaxLik(pIMf,pIMg,pIMh,pIMp,1,inConType);
    inMeasures(&flLik,ML,pIMIdeal,pIMf,pIMg,pIMh,pIMp,inConType,OK);
         }
  else if (strcmp("MAP",av[6]) == 0)
         {
        if (strcmp("LINEAR",av[7]) == 0)
      inConType = LINEAR;
    else if (strcmp("CIRCULAR",av[7]) == 0)
```

```
            inConType = CIRCULAR;
             else
            inStatus = NK;

          inIMAMAP(pIMf,pIMg,pIMh,pIMp,1,inConType);
          inMeasures(&flLik,MAP,pIMIdeal,pIMf,pIMg,pIMh,pIMp,inConType,OK);
               }
           else
          inStatus = NK;

               if (inCount1 == 0)
               {
          flMaxLik = flLik;
          inIMACopyIma(pIMBest,pIMf);
               }
               else
               {
          if (flMaxLik < flLik)
              {
            flMaxLik = flLik;
            inIMACopyIma(pIMBest,pIMf);
              }
               }

        inIMACopyIma(pIMp,pIMf);
               inUpStep--;
        inRepStep--;
         }

      sprintf(szBuffer,"best-%s.ima",av[1]);
      inIMASaveIma(szBuffer,pIMBest);
      sprintf(szBuffer,"%s.ima",av[1]);
      inIMASaveIma(szBuffer,pIMf);
       }
  }

  if (pIMIdeal != NULL)
    inIMAFreeImaMem(pIMIdeal);
  if (pIMBaseg != NULL)
      inIMAFreeImaMem(pIMBaseg);
  if (pIMg != NULL)
      inIMAFreeImaMem(pIMg);
  if (pIMp != NULL)
    inIMAFreeImaMem(pIMp);
  if (pIMBaseh != NULL)
      inIMAFreeImaMem(pIMBaseh);
  if (pIMh != NULL)
    inIMAFreeImaMem(pIMh);
  if (pIMf != NULL)
    inIMAFreeImaMem(pIMf);
  if (pIMBest != NULL)
    inIMAFreeImaMem(pIMBest);

  return(inStatus);
}

int inMeasures(float  *pflLik, int inAlgTyp, IMAGE *pIMIdeal, IMAGE *pIMf, IMAGE *pIMg,
                        IMAGE *pIMh, IMAGE *pIMp, int inConType, int inFlag)
{
  int                          inStatus,
                                      inRowSize,
                                      inColSize;
  float                        flLik1,
                                      flLik2,
                                      flRes,
                                      flLPN,
                                      flCor,
                                      flLoIdEn,
                                      flHiIdEn,
                                      flLoReEn,
                                      flHiReEn,
                                      flToSi,
                                      flLoSi,
                                      flHiSi,
                                      flVA;
  IMAGE                    *pIMIma1,
```

```
                                *pIMIma2,
                                *pIMIma3,
                                *pIMIma4,
                                *pIMIma5;

inStatus = OK;

if (pIMIdeal != NULL)
{
  inRowSize = pIMIdeal->inNumRows;
      inColSize = pIMIdeal->inNumCols;
}
else
{
  inRowSize = pIMg->inNumRows;
      inColSize = pIMg->inNumCols;
}

pIMIma1 = pIMIMAAskImaMem(inRowSize,inColSize);
inIMAResample(pIMIma1,pIMf);

pIMIma2 = pIMIMAAskImaMem(inRowSize,inColSize);
inIMAResample(pIMIma2,pIMg);

pIMIma3 = pIMIMAAskImaMem(pIMh->inNumRows,pIMh->inNumRows);
inIMAImaConvolution(pIMIma3,pIMh,pIMf,inConType);
pIMIma4 = pIMIMAAskImaMem(pIMh->inNumRows,pIMh->inNumRows);
inIMAImaFunOp(pIMIma4,pIMIma3,IMRA);
inIMAFreeImaMem(pIMIma3);
pIMIma3 = pIMIMAAskImaMem(inRowSize,inColSize);
inIMAResample(pIMIma3,pIMIma4);
inIMAFreeImaMem(pIMIma4);

pIMIma4 = pIMIMAAskImaMem(inRowSize,inColSize);
inIMAResample(pIMIma4,pIMh);

pIMIma5 = pIMIMAAskImaMem(inRowSize,inColSize);
inIMAResample(pIMIma5,pIMp);

if (inFlag == NK)
{
  flLik1 = 0;
  flLik2 = 0;
  flRes = 0;
}
else
{
  inIMALikelihood(&flLik1,pIMIma2,pIMIma3,pIMIma1,pIMIma5,ML);
  inIMALikelihood(&flLik2,pIMIma2,pIMIma3,pIMIma1,pIMIma5,MAP);
      inIMAImaMetric(&flRes,pIMIma2,pIMIma3,LPNORM,2);
}

if (pIMIdeal == NULL)
{
  flLPN = 0;
      flCor = 0;
      flLoIdEn = 0;
      flHiIdEn = 0;
      flLoReEn = 0;
      flHiReEn = 0;
      flToSi = 0;
      flLoSi = 0;
      flHiSi = 0;
      flVA = 0;
}
else
{
      inIMAImaMetric(&flLPN,pIMIdeal,pIMIma1,LPNORM,2);
  inIMAImaMetric(&flCor,pIMIdeal,pIMIma1,CORRELATION,0);
}

inIMAFreeImaMem(pIMIma1);
inIMAFreeImaMem(pIMIma2);
inIMAFreeImaMem(pIMIma3);
inIMAFreeImaMem(pIMIma4);
inIMAFreeImaMem(pIMIma5);
```

```
  printf("%+e,%+e,%+e,%+e,%+e,%+e,%+e,%+e,%+e,%+e,%+e\n",
            flLik1,flLik2,flRes,flLPN,flCor);

  if (inAlgTyp == ML)
    (*pflLik) = flLik1;
  else if (inAlgTyp == MAP)
    (*pflLik) = flLik2;

  return(inStatus);
}
```

```
NAME
    threshold.cpp
USAGE
    threshold szOutIma szInIma szDirection flThreshold
INPUT PARAMETERS
    string              szOutIma
    string              szInIma
    string              szDirection
    float               flThreshold
DESCRIPTION
    Thresholds the image stored in the file <szInIma>. If <szDirection> is UP, all pixels with a
    value greater than <flThreshold> are assigned a value equal to <flThreshold>. If
    <szDirection> is DOWN is the reverse. The resulting image is stored in the file <szOutIma>.
PROGRAMMER
    Pablo Zegers
```

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "..\..\Tools\global\global.h"
#include "..\..\Tools\image\image.h"
#include "..\..\Tools\mcomplex\mcomplex.h"
#include "..\..\Tools\dft\dft.h"

#define UP                              0
#define DOWN                    1

int main(int ac, char *av[])
{
  char                                  szBuffer[100];
  int                                   inStatus,
                                                inDirection,
                                                inCount1,
                                                inCount2;
  float                                 flThreshold,
                                                flImaginary;
  mcomplex                              mcValue;
  IMAGE                                *pIMIma1,
                                               *pIMIma2;


  inStatus = OK;

  pIMIma1 = NULL;
  pIMIma2 = NULL;

  if (ac != 5)
  {
    printf("USAGE: threshold szOutIma szInIma szDirection flThreshold\n");
    inStatus = NK;
  }
  else
  {
    sprintf(szBuffer,"%s.ima",av[2]);
    pIMIma1 = pIMIMAReadIma(szBuffer);

        if (pIMIma1 == NULL)
        {
      printf("The image %s could not be read!\n",av[2]);
      inStatus = NK;
        }
        else
        {
      if (strcmp(av[3],"UP") == 0)
        inDirection = UP;
          else if (strcmp(av[3],"DOWN") == 0)
        inDirection = DOWN;
          else
                inDirection = MERROR;

          if (inDirection == MERROR)
          {
        inStatus = NK;
          }
          else
          {
```

```
            flThreshold = (float)atof(av[4]);

        pIMIma2 = pIMIMAAskImaMem(pIMIma1->inNumRows,pIMIma1->inNumCols);

            for (inCount1=0;inCount1<pIMIma1->inNumRows;inCount1++)
                {
                  for (inCount2=0;inCount2<pIMIma1->inNumCols;inCount2++)
                  {
                    inIMAGetImaElem(&mcValue,pIMIma1,inCount1,inCount2);

                        if (inDirection == UP)
                        {
                          if (mcValue.flRe > flThreshold)
                          {
                                flImaginary = mcValue.flIm;
                                voSetCom(&mcValue,flThreshold,flImaginary);
                          }
                          inIMASetImaElem(pIMIma2,inCount1,inCount2,mcValue);
                        }
                        else
                        {
                          if (mcValue.flRe < flThreshold)
                          {
                                flImaginary = mcValue.flIm;
                                voSetCom(&mcValue,flThreshold,flImaginary);
                          }
                          inIMASetImaElem(pIMIma2,inCount1,inCount2,mcValue);
                        }
                  }
                }

        sprintf(szBuffer,"%s.ima",av[1]);
        inIMASaveIma(szBuffer,pIMIma2);
          }
        }
  }

  if (pIMIma1 != NULL)
    inIMAFreeImaMem(pIMIma1);
  if (pIMIma2 != NULL)
    inIMAFreeImaMem(pIMIma2);

  return(inStatus);
}
```

```
NAME
    totalflip.cpp
USAGE
    totalflip szOutIma szInIma
INPUT PARAMETERS
    string              szOutIma
    string              szInIma
DESCRIPTION
    Flips the image stored in the file <szInIma> one time horizontally, and one time vertically.
    The resulting image is stored in the file <szOutIma>.
PROGRAMMER
    Pablo Zegers
```

```
#include <stdlib.h>
#include <stdio.h>
#include "..\..\Tools\global\global.h"
#include "..\..\Tools\image\image.h"
#include "..\..\Tools\mcomplex\mcomplex.h"
#include "..\..\Tools\dft\dft.h"

int main(int ac, char *av[])
{
  char                                szBuffer[100];
  int                                 inStatus;
  IMAGE                             *pIMIma1,
                                        *pIMIma2,
                                        *pIMIma3;

  inStatus = OK;

  pIMIma1 = NULL;
  pIMIma2 = NULL;
  pIMIma3 = NULL;

  if (ac != 3)
  {
    printf("USAGE: totalflip szOutIma szInIma\n");
    inStatus = NK;
  }
  else
  {
    sprintf(szBuffer,"%s.ima",av[2]);
    pIMIma1 = pIMIMAReadIma(szBuffer);

        if (pIMIma1 == NULL)
        {
      printf("The image %s could not be read!\n",av[2]);
      inStatus = NK;
        }
        else
        {
      pIMIma2 = pIMIMAAskImaMem(2*pIMIma1->inNumRows,2*pIMIma1->inNumCols);
          inIMATotalFlip(pIMIma2,pIMIma1);

      pIMIma3 = pIMIMAAskImaMem(pIMIma2->inNumRows,pIMIma2->inNumCols);
          inIMAImaFunOp(pIMIma3,pIMIma2,IMRA);

      sprintf(szBuffer,"%s.ima",av[1]);
      inIMASaveIma(szBuffer,pIMIma3);
        }
  }

  if (pIMIma1 != NULL)
    inIMAFreeImaMem(pIMIma1);
  if (pIMIma2 != NULL)
    inIMAFreeImaMem(pIMIma2);
  if (pIMIma3 != NULL)
    inIMAFreeImaMem(pIMIma3);

  return(inStatus);
}
```

```
3NAME
    dft.h
DESCRIPTION
    Header file for dft.cpp
PROGRAMMER
    Pablo Zegers
```

```
//-------------------------------------------------------------------------------------------------
-------------------------
#ifndef FNDFT
//-------------------------------------------------------------------------------------------------
-------------------------

//-------------------------------------------------------------------------------------------------
-------------------------
// INCLUDE FILES
//-------------------------------------------------------------------------------------------------
-------------------------
#include "..\Global\Global.h"
#include "..\mcomplex\mcomplex.h"

//-------------------------------------------------------------------------------------------------
-------------------------
// CONSTANTS
//-------------------------------------------------------------------------------------------------
-------------------------
#define DFT                             -1
#define IDFT                    +1

//-------------------------------------------------------------------------------------------------
-------------------------
// PUBLIC PROTOTYPES
//-------------------------------------------------------------------------------------------------
-------------------------
void voDFTNDFT(float *pflData, unsigned long *pulDims, int inNumDim, int inOpTy);

//-------------------------------------------------------------------------------------------------
-------------------------
#define FNDFT                   0
#endif
//-------------------------------------------------------------------------------------------------
-------------------------
```

```
NAME
    dft.cpp
DESCRIPTION
    Routines needed to calculate the FFTs.
PROGRAMMER
    Pablo Zegers
```

```
//-------------------------------------------------------------------------------------------------------------
--------------------------
// DESCRIPTION : Functions to perform FTs in n-dimensions arrays
// AUTHOR      : Pablo Zegers
// DATE           : February, 1999
//-------------------------------------------------------------------------------------------------------------
--------------------------

//-------------------------------------------------------------------------------------------------------------
--------------------------
// INCLUDE FILES
//-------------------------------------------------------------------------------------------------------------
--------------------------
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <malloc.h>
#include <math.h>

#include "dft.h"

//-------------------------------------------------------------------------------------------------------------
--------------------------
// CONSTANTS
//-------------------------------------------------------------------------------------------------------------
--------------------------
#define SWAP(a,b)      tempr=(a);(a)=(b);(b)=tempr

//-------------------------------------------------------------------------------------------------------------
--------------------------
// INCLUDE FILES
//-------------------------------------------------------------------------------------------------------------
--------------------------
int inIsPowerOf2(int inNumber);
void voNDFFT(float *pflData, unsigned long *pulDims, int inNumDim, int inOpTy);
void voNDFT(float *pflData, unsigned long *pulDims, int inNumDim, int inOpTy);
inInitializeIndex(int *pinVecInd, int inNumInd);
inIncrementIndex(int *pinVecInd, unsigned long *pulDims, int inNumInd);

//-------------------------------------------------------------------------------------------------------------
--------------------------
// PUBLIC ROUTINES
//-------------------------------------------------------------------------------------------------------------
--------------------------
void voDFTNDFT(float *pflData, unsigned long *pulDims, int inNumDim, int inOpTy)
{
  int                          inFlag,
                                      inCount1,
                                      ulTotalValues;

  inFlag = YES;
  for (inCount1=0;inCount1<inNumDim;inCount1++)
  {
    if (inIsPowerOf2(pulDims[inCount1]) != YES)
      inFlag = NO;
  }

  if (inFlag == YES)
    voNDFFT(pflData,pulDims,inNumDim,inOpTy);
  else
    voNDFT(pflData,pulDims,inNumDim,inOpTy);

  if (inOpTy == IDFT)
  {
    ulTotalValues = 1;
    for (inCount1=0;inCount1<inNumDim;inCount1++)
      ulTotalValues *= pulDims[inCount1];
```

```
    for (inCount1=0;inCount1<2*ulTotalValues;inCount1++)
      pflData[inCount1] /= ulTotalValues;
  }
}

//-------------------------------------------------------------------------------------------------
-------------------------
// PRIVATE ROUTINES
//-------------------------------------------------------------------------------------------------
-------------------------

//-------------------------------------------------------------------------------------------------
-------------------------
//-------------------------------------------------------------------------------------------------
-------------------------
int inIsPowerOf2(int inNumber)
{
  float               flResult;

  flResult = (float)inNumber;

  while (flResult > 1)
    flResult /= 2;

  if (flResult != 1)
    return(NO);
  else
    return(YES);
}

//-------------------------------------------------------------------------------------------------
-------------------------
//-------------------------------------------------------------------------------------------------
-------------------------
void voNDFFT(float *pflData, unsigned long *pulDims, int inNumDim, int inOpTy)
{
  int                         ndim,
                                      isign,
                                      idim;
  unsigned long               i1,
                                      i2,
                                      i3,
                                      i2rev,
                                      i3rev,
                                      ip1,
                                      ip2,
                                      ip3,
                                      ifp1,
                                      ifp2,
                                      ibit,
                                      k1,
                                      k2,
                                      n,
                                      nprev,
                                      nrem,
                                      ntot;
  unsigned long    *nn;
  float                     *data;
  float                       tempi,
                                      tempr,
                                      theta,
                                      wi,
                                      wpi,
                                      wpr,
                                      wr,
                                      wtemp;

  // modification to the original algorithm
  data = pflData-1;
  nn = pulDims-1;
  ndim = inNumDim;
  isign = inOpTy;

  for (ntot=1,idim=1;idim<=ndim;idim++)
    ntot *= nn[idim];
```

```
  nprev = 1;

  for (idim=ndim;idim>=1;idim--)
  {
    n = nn[idim];
        nrem = ntot/(n*nprev);
        ip1 = nprev<<1;
        ip2 = ip1*n;
        ip3 = ip2*nrem;
        i2rev = 1;

        for (i2=1;i2<=ip2;i2+=ip1)
    {
      if (i2 < i2rev)
      {
        for (i1=i2;i1<=i2+ip1-2;i1+=2)
        {
                  for (i3=i1;i3<=ip3;i3+=ip2)
                  {
                    i3rev = i2rev+i3-i2;
                        SWAP(data[i3],data[i3rev]);
                        SWAP(data[i3+1],data[i3rev+1]);
                  }
                }
        }

        ibit = ip2 >> 1;
        while ((ibit >= ip1) && (i2rev > ibit))
        {
          i2rev -= ibit;
              ibit >>= 1;
        }

        i2rev += ibit;
      }

      ifp1 = ip1;
      while (ifp1 < ip2)
      {
        ifp2 = ifp1 << 1;
        theta = ((float)(isign*6.28318530717959/(ifp2/ip1)));
        wtemp = (float)sin(0.5*theta);
        wpr = ((float)(-2.0*wtemp*wtemp));
        wpi = (float)sin(theta);
        wr = 1.0;
        wi = 0.0;
        for (i3=1;i3<=ifp1;i3+=ip1)
        {
          for (i1=i3;i1<=i3+ip1-2;i1+=2)
              {
                for (i2=i1;i2<=ip3;i2+=ifp2)
                {
                  k1 = i2;
                      k2 = k1+ifp1;
                      tempr = (float)wr*data[k2]-(float)wi*data[k2+1];
                      tempi = (float)wr*data[k2+1]+(float)wi*data[k2];
                      data[k2] = data[k1]-tempr;
                      data[k2+1] = data[k1+1]-tempi;
                      data[k1] += tempr;
                      data[k1+1] += tempi;
                }
              }

              wr = (wtemp = wr)*wpr-wi*wpi+wr;
              wi = wi*wpr+wtemp*wpi+wi;
        }

        ifp1 = ifp2;
      }

      nprev *= n;
  }
}

//-------------------------------------------------------------------------------------------------
--------------------------
```

```
//-------------------------------------------------------------------------------------------------
-------------------------
void voNDFT(float *pflData, unsigned long *pulDims, int inNumDim, int inOpTy)
{
  int                    *pinDaInd,
                             *pinReInd;
  int                    inCount1,
                             inCount2;
  unsigned long          ulTotalValues,
                             ulPointPos1,
                             ulPointPos2;
  float                  *pflResult;
  float                  flBuffer;
  mcomplex               mcBuffer1,
                             mcBuffer2,
                             mcBuffer3,
                             mcBuffer4;

  // calculates the number of real values needed to store the resulting matrix
  ulTotalValues = 1;
  for (inCount1=0;inCount1<inNumDim;inCount1++)
    ulTotalValues *= pulDims[inCount1];

  // asks memory to store the vector of data indexes
  pinDaInd = (int *) calloc(inNumDim,sizeof(int));

  // asks memory to store the vector of result indexes
  pinReInd = (int *) calloc(inNumDim,sizeof(int));

  // asks memory to store the results
  pflResult = (float *) calloc(2*ulTotalValues,sizeof(float));

  inInitializeIndex(pinReInd,inNumDim);
  ulPointPos1 = 0;
  do
  {
    voSetCom(&mcBuffer1,0,0);
    inInitializeIndex(pinDaInd,inNumDim);
    ulPointPos2 = 0;
    do
       {
      flBuffer = 0;
      for (inCount2=0;inCount2<inNumDim;inCount2++)
        flBuffer +=
((float)pinReInd[inCount2])*((float)pinDaInd[inCount2])/((float)pulDims[inCount2]);
      voSetCom(&mcBuffer2,0,(float)(inOpTy*2*PI*flBuffer));
      voExpCom(&mcBuffer3,mcBuffer2);
      voSetCom(&mcBuffer2,pflData[2*ulPointPos2],pflData[2*ulPointPos2+1]);
      voMulCom(&mcBuffer4,mcBuffer2,mcBuffer3);
      voAssignCom(&mcBuffer2,mcBuffer1);
      voAddCom(&mcBuffer1,mcBuffer2,mcBuffer4);

      inIncrementIndex(pinDaInd,pulDims,inNumDim);
      ulPointPos2++;

       } while (ulPointPos2 < ulTotalValues);

    pflResult[2*ulPointPos1] = mcBuffer1.flRe;
    pflResult[2*ulPointPos1+1] = mcBuffer1.flIm;

    inIncrementIndex(pinReInd,pulDims,inNumDim);
    ulPointPos1++;

  } while (ulPointPos1 < ulTotalValues);

  // copies the result vector into the data vector
  for (ulPointPos1=0;ulPointPos1<2*ulTotalValues;ulPointPos1++)
    pflData[ulPointPos1] = pflResult[ulPointPos1];

  // frees the memory used to store the data indexes
  free(pinDaInd);
  // frees the memory used to store the result indexes
  free(pinReInd);
  // frees the memory used to store the results
  free(pflResult);
}
```

```
//-------------------------------------------------------------------------------------------------
-------------------------
//-------------------------------------------------------------------------------------------------
-------------------------
inInitializeIndex(int *pinVecInd, int inNumInd)
{
  int                           inCount1;

  for (inCount1=0;inCount1<inNumInd;inCount1++)
    pinVecInd[inCount1] = 0;

  return(OK);
}

//-------------------------------------------------------------------------------------------------
-------------------------
//-------------------------------------------------------------------------------------------------
-------------------------
inIncrementIndex(int *pinVecInd, unsigned long *pulDims, int inNumInd)
{
  int                           inFlag,
                                       inCount1;

  inFlag = YES;
  for (inCount1=inNumInd-1;inCount1>=0;inCount1--)
  {
    if (inFlag == YES)
          pinVecInd[inCount1] +=1;

    if (pinVecInd[inCount1] == ((int)pulDims[inCount1]))
    {
      inFlag = YES;
          pinVecInd[inCount1] = 0;
       }
       else
       {
      inFlag = NO;
        }
  }

  return(OK);
}
```

```
NAME
    pgm.h
DESCRIPTION
    Header file for pgm.cpp
PROGRAMMER
    Pablo Zegers
```

```
//-----------------------------------------------------------------------------
#ifndef FPGM
//-----------------------------------------------------------------------------

//-----------------------------------------------------------------------------
// INCLUDE FILES
//-----------------------------------------------------------------------------
#include "..\Global\Global.h"
#include "..\mcomplex\mcomplex.h"
#include "..\Image\Image.h"

//-----------------------------------------------------------------------------
// CONSTANTS DEFINITIONS
//-----------------------------------------------------------------------------
#define P2                      0
#define P5                      1
#define MAXVAL          255

//-----------------------------------------------------------------------------
// PUBLIC PROTOTYPES
//-----------------------------------------------------------------------------
IMAGE *pIMFILReadPGM(char *szPGMFile);
int inFILSavePGM(char *szPGMFile, IMAGE *pIMIma, int inMaxVal);

//-----------------------------------------------------------------------------
#define FPGM 0
#endif
//-----------------------------------------------------------------------------
```

NAME
pgm.cpp
DESCRIPTION
Routines to translate an image from the PGM format to the IMA format, and vice versa.
PROGRAMMER
Pablo Zegers

```
//-------------------------------------------------------------------------
// DESCRIPTION : Functions to manipulate PGM images
// AUTHOR      : Pablo Zegers
// DATE        : October, 1997
//-------------------------------------------------------------------------

//-------------------------------------------------------------------------
// INCLUDE FILES
//-------------------------------------------------------------------------
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <malloc.h>
#include <math.h>
#include <io.h>
#include <fcntl.h>
#include <sys/stat.h>
#include "pgm.h"

//-------------------------------------------------------------------------
// PUBLIC ROUTINES
//-------------------------------------------------------------------------

//-------------------------------------------------------------------------
//-------------------------------------------------------------------------
IMAGE *pIMFILReadPGM(char *szPGMFile)
{
  unsigned char              ucBuffer;
  char                       szBuffer[MSTRING],
                                    szMagic[MSTRING];
  int                        inStatus,
                                    inHandle,
                                    inNumRows,
                                    inNumCols,
                                    inMaxVal,
                                    inCount1,
                                    inCount2;
  IMAGE                  *pIMIma;

  inStatus = OK;

  pIMIma = NULL;

  inHandle = open(szPGMFile,O_RDONLY|O_BINARY);
  if (inHandle < 0)
  {
    printf("FILE %s COUDN'T BE OPENED.\n",szPGMFile);
    inStatus = NK;
  }
  else
  {
    read(inHandle,(void*)szBuffer,15*sizeof(char));
    szBuffer[13] = 0;

    sscanf(szBuffer,"%s%d%d%d",szMagic,&inNumCols,&inNumRows,&inMaxVal);

    if ((strcmp(szMagic,"P5") != 0))
        {
      printf("WRONG FILE FORMAT.\n");
      inStatus = NK;
       }
       else
       {
      pIMIma = pIMIMAAskImaMem(inNumRows,inNumCols);

         if (pIMIma == NULL)
         {
           inStatus = NK;
```

```
        }
        else
        {
          for (inCount1=0;inCount1<inNumRows;inCount1++)
              {
         for (inCount2=0;inCount2<inNumCols;inCount2++)
              {
           read(inHandle,(void*)&ucBuffer,sizeof(char));
           voSetCom(&(pIMIma->ppmcElem[inCount1][inCount2]),((float)((int)(ucBuffer))),0);
              }
            }
        }
      }

    close(inHandle);
  }

  return(pIMIma);
}

//-----------------------------------------------------------------------------
//-----------------------------------------------------------------------------
int inFILSavePGM(char *szPGMFile, IMAGE *pIMIma, int inMaxVal)
{
  char                          szBuffer[MSTRING];
  int                           inStatus,
                                        inHandle,
inValues,
                                        inCount1,
                                        inCount2;
  FILE                    *pFIStream;

  inStatus = OK;

//  inHandle = creat(szPGMFile,_S_IREAD|_S_IWRITE);
pFIStream = fopen(szPGMFile,"w");
  if (inHandle == -1)
  {
    printf("IT CAN'T OPEN FILE %s\n",szPGMFile);
    inStatus = NK;
  }
  else
  {
fprintf(pFIStream,"%s\n%d\n%d\n%d\n","P2",pIMIma->inNumCols,pIMIma->inNumRows,inMaxVal);
//    sprintf(szBuffer,"%s\n%d\n%d\n%d\n","P2",pIMIma->inNumCols,pIMIma->inNumRows,inMaxVal);
//     write(inHandle,(void *)szBuffer,strlen(szBuffer));

inValues = 0;
    for (inCount1=0;inCount1<pIMIma->inNumRows;inCount1++)
        {
      for (inCount2=0;inCount2<pIMIma->inNumCols;inCount2++)
          {
if (inValues < 40)
{
  inValues++;
}
else
{
  inValues = 0;
  fprintf(pFIStream,"\n");
}
fprintf(pFIStream,"%i ",(unsigned char)(floor(pIMIma->ppmcElem[inCount1][inCount2].flRe)));
//        szBuffer[0] = (unsigned char)(floor(pIMIma->ppmcElem[inCount1][inCount2].flRe));
//        write(inHandle,(void *)szBuffer,1);
          }
        }
  }

//  close(inHandle);
fclose(pFIStream);

  return(inStatus);
}
```

```
NAME
    global.h
DESCRIPTION
    Defines all the global constants.
PROGRAMMER
    Pablo Zegers
```

```
//---------------------------------------------------------------------
// DEFINITIONS
//---------------------------------------------------------------------
#define OK                      0
#define NK                      -1

#define YES                     0
#define NO                      -1

#define MSTRING                 100
#define MERROR          -1

#define IPORT           0
#define OPORT           1
#define NEURON          2
#define SYSTEM          3

#define PI                      3.1415926

#define PRECISION       6
```

```
NAME
    image.h
DESCRIPTION
    Header file for image.cpp
PROGRAMMER
    Pablo Zegers
```

```
//-------------------------------------------------------------------------------------------------
-------------------------
#ifndef FIMAGE
//-------------------------------------------------------------------------------------------------
-------------------------

//-------------------------------------------------------------------------------------------------
-------------------------
// INCLUDE FILES
//-------------------------------------------------------------------------------------------------
-------------------------
#include "..\Global\Global.h"
#include "..\mcomplex\mcomplex.h"
#include "..\dft\dft.h"

//-------------------------------------------------------------------------------------------------
-------------------------
// CONSTANTS DEFINITIONS
//-------------------------------------------------------------------------------------------------
-------------------------
#define ADD                             0
#define SUB                             1
#define MUL                             2
#define DIV                             3

#define REAL                    0
#define IMAGINARY               1
#define ZERE                    2
#define ZEIM                    3
#define IMRA                    4
#define MAGNITUDE               5
#define PHASE                   6
#define EXP                             7
#define LOG                             8
#define POW                             9

#define LPNORM                  0
#define CORRELATION             1

#define CORNERED                0
#define CENTERED                1
#define SPREAD                  2

#define RECTANGULAR             0
#define HAMMING                         1
#define HANNING                         2
#define BLACKMAN                3

#define LINEAR                  0
#define CIRCULAR                1

#define CLS                             0
#define ML                              1
#define MAP                             2

//-------------------------------------------------------------------------------------------------
-------------------------
// STRUCTURE DEFINITIONS
//-------------------------------------------------------------------------------------------------
-------------------------
typedef struct
{
  int                                   inIndex,
                                                inNumRows,
                                                inNumCols;

  mcomplex                      **ppmcElem;
} IMAGE;
```

```
//-------------------------------------------------------------------------------------------------
--------------------------
// PUBLIC PROTOTYPES
//-------------------------------------------------------------------------------------------------
--------------------------
IMAGE *pIMIMAAskImaMem(int inNumRows, int inNumCols);
int inIMAFreeImaMem(IMAGE *pIMIma);
int inIMASetImaIndex(IMAGE *pIMIma, int inIndex);
int inIMAGetImaIndex(int *pinIndex, IMAGE *pIMIma);
int inIMASetImaElem(IMAGE *pIMIma, int inRow, int inCol, mcomplex mcValue);
int inIMAGetImaElem(mcomplex *pmcValue, IMAGE *pIMIma, int inRow, int inCol);
IMAGE *pIMIMAReadIma(char *szName);
int inIMASaveIma(char *szName, IMAGE *pIMIma);
int inImaMax(float *pflMax, IMAGE *pIMIma, int inType);
int inImaMin(float *pflMin, IMAGE *pIMIma, int inType);
int inIMAImaMean(mcomplex *pmcMean, IMAGE *pIMIma);
int inIMAImaVar(mcomplex *pmcVar, IMAGE *pIMIma);
int inIMAInsertIma(IMAGE *pIMIma, IMAGE *pIMSub, int inRowIn, int inColIn);
int inIMAExtractIma(IMAGE *pIMSub, IMAGE *pIMIma, int inRowIn, int inColIn, int inDelRows, int
inDelCols);
int inIMACopyIma(IMAGE *pIMDesIma, IMAGE *pIMOriIma);
int inIMAHorFlipIma(IMAGE *pIMOutIma, IMAGE *pIMInIma);
int inIMAVerFlipIma(IMAGE *pIMOutIma, IMAGE *pIMInIma);
int inIMAResample(IMAGE *pIMOutIma, IMAGE *pIMInIma);
int inIMAImaScaOp(IMAGE *pIMImaB, IMAGE *pIMImaA, mcomplex mcValue, int inOpType);
int inIMAImaMatOp(IMAGE *pIMImaC, IMAGE *pIMImaA, IMAGE *pIMImaB, int inOpType);
int inIMAImaFunOp(IMAGE *pIMImaB, IMAGE *pIMImaA, int inOpType);
int inIMADFTIma(IMAGE *pIMOutIma, IMAGE *pIMInIma, int inTransformType);
int inIMAIsPowerOf2(int inNumber);
int inIMAGetNextPowerOf2(int inNumber);
int inIMACLS(IMAGE *pIMf, IMAGE *pIMg, IMAGE *pIMh, float flGamma);
int inIMAMaxLik(IMAGE *pIMf, IMAGE *pIMg, IMAGE *pIMh, IMAGE *pIMPrior, int inNumIter,
                                int inType);
int inIMAMAP(IMAGE *pIMf, IMAGE *pIMg, IMAGE *pIMh, IMAGE *pIMPrior, int inNumIter,
                        int inType);
int inIMAElipsoidalAperture(IMAGE *pIMAperture, int inAxis1, int inAxis2);
int inIMASlantedStripAperture(IMAGE *pIMAperture, int inLength, int inWidth, float flAngle);
int inIMAImaConvolution(IMAGE *pIMImaC, IMAGE *pIMImaA, IMAGE *pIMImaB, int inConType);
int inIMAImaCorrelation(IMAGE *pIMImaC, IMAGE *pIMImaA, IMAGE *pIMImaB, int inCorType);
int inIMAOTFFromAperture(IMAGE *pIMOTF, IMAGE *pIMAperture, int inConType);
int inIMAPSFFromAperture(IMAGE *pIMPSF, IMAGE *pIMAperture, int inConType);
int inIMAStretch(IMAGE *pIMOutIma, IMAGE *pIMInIma, int inMinVal, int inMaxVal);
int inIMAImaMetric(float *pflDistance, IMAGE *pIMA, IMAGE *pIMB, int inMetricType, float
flParam);
int inIMAImaShift(IMAGE *pIMImaOut, IMAGE *pIMImaIn);
int inIMASimMap(IMAGE *pIMMap, IMAGE *pIMRef, IMAGE *pIMIma, int inBlockSize, int inMetricType,
float flParam);
int inIMAAverageSimilarity(float *pflAveSim, int inLimit, IMAGE *pIMIdeal, IMAGE *pIMImage);
int inIMAZeroPadImage(IMAGE *pIMOutIma, IMAGE *pIMInIma, int inType);
int inIMAGetWindow(IMAGE *pIMWindow, int inWindowType);
int inIMALikelihood(float *pflLik, IMAGE *pIMg, IMAGE *pIMEstBlur, IMAGE *pIMf,
                                IMAGE *pIMPrior, int inType);
int inIMATotalFlip(IMAGE *pIMOutIma, IMAGE *pIMInIma);

//-------------------------------------------------------------------------------------------------
--------------------------
#define FIMAGE 0
#endif
//-------------------------------------------------------------------------------------------------
--------------------------
```

```
NAME
    image.cpp
DESCRIPTION
    Routines needed to manipulate an image.
PROGRAMMER
    Pablo Zegers
```

```
//-------------------------------------------------------------------------------------------------
// DESCRIPTION : Functions for manipulating images
// AUTHOR      : Pablo Zegers
// DATE        : March, 1999
//-------------------------------------------------------------------------------------------------

//-------------------------------------------------------------------------------------------------
// INCLUDE FILES
//-------------------------------------------------------------------------------------------------
#include <stdlib.h>
#include <stdio.h>
#include <io.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <string.h>
#include <malloc.h>
#include <math.h>
#include "Image.h"

//-------------------------------------------------------------------------------------------------
// CONSTANTS
//-------------------------------------------------------------------------------------------------
#define ENERGYTHRESHOLD              (float)0.000001

//-------------------------------------------------------------------------------------------------
// PRIVATE PROTOTYPES
//-------------------------------------------------------------------------------------------------
IMAGE *pIMAskImaMem(int inNumRows, int inNumCols);
int inFreeImaMem(IMAGE *pIMIma);
int inImaMin(float *pflMin, IMAGE *pIMIma, int inType);
int inImaMax(float *pflMax, IMAGE *pIMIma, int inType);
int inImaMean(mcomplex *pmcMean, IMAGE *pIMIma);
int inImaVar(mcomplex *pmcVar, IMAGE *pIMIma);
int inInsertIma(IMAGE *pIMIma, IMAGE *pIMSub, int inRowIn, int inColIn);
int inIsWithinScope(IMAGE *pIMIma, int inRow, int inCol);
int inExtractIma(IMAGE *pIMSub, IMAGE *pIMIma, int inRowIn, int inColIn,
                 int inDelRows, int inDelCols);
int inCopyIma(IMAGE *pIMDesIma, IMAGE *pIMOriIma);
int inHorFlipIma(IMAGE *pIMOutIma, IMAGE *pIMInIma);
int inVerFlipIma(IMAGE *pIMOutIma, IMAGE *pIMInIma);
int inImaScaOp(IMAGE *pIMImaB, IMAGE *pIMImaA, mcomplex mcValue,
               int inOpType);
int inImaMatOp(IMAGE *pIMImaC, IMAGE *pIMImaA, IMAGE *pIMImaB, int inOpType);
int inImaFunOp(IMAGE *pIMImaB, IMAGE *pIMImaA, int inOpType);
int inDFTIma(IMAGE *pIMOutIma, IMAGE *pIMInIma, int inTransformType);
int inImaConvolution(IMAGE *pIMImaC, IMAGE *pIMImaA, IMAGE *pIMImaB, int inConType);
int inImaCorrelation(IMAGE *pIMImaC, IMAGE *pIMImaA, IMAGE *pIMImaB, int inCorType);
int inOTFFromAperture(IMAGE *pIMOTF, IMAGE *pIMAperture, int inCorTyp);
int inImaMetric(float *pflDistance, IMAGE *pIMA, IMAGE *pIMB, int inMetricType, float flParam);
int inImaShift(IMAGE *pIMImaOut, IMAGE *pIMImaIn);
int inSimMap(IMAGE *pIMMap, IMAGE *pIMRef, IMAGE *pIMIma, int inBlockSize, int inMetricType,
float flParam);
int inGetAveSim(float *pflAveSim, IMAGE *pIMSimMap, float flRadius, float flWidth,
                IMAGE *pIMSpectrum);
int inZeroPadImage(IMAGE *pIMOutIma, IMAGE *pIMInIma, int inType);
int inGetWindow(IMAGE *pIMWindow, int inWindowType);
int inLikelihood(float *pflLik, IMAGE *pIMg, IMAGE *pIMEstBlur, IMAGE *pIMf,
                 IMAGE *pIMPrior, int inType);
float flLogNatFac(float flNumber);
int inTotalFlip(IMAGE *pIMOutIma, IMAGE *pIMInIma);
```

```
//-------------------------------------------------------------------------------------------------
// PUBLIC ROUTINES
//-------------------------------------------------------------------------------------------------

//-------------------------------------------------------------------------------------------------
//-------------------------------------------------------------------------------------------------
IMAGE *pIMIMAAskImaMem(int inNumRows, int inNumCols)
{
  return(pIMAskImaMem(inNumRows,inNumCols));
}

//-------------------------------------------------------------------------------------------------
//-------------------------------------------------------------------------------------------------
int inIMAFreeImaMem(IMAGE *pIMIma)
{
  return(inFreeImaMem(pIMIma));
}

//-------------------------------------------------------------------------------------------------
//-------------------------------------------------------------------------------------------------
int inIMASetImaIndex(IMAGE *pIMIma, int inIndex)
{
  pIMIma->inIndex = inIndex;

  return(OK);
}

//-------------------------------------------------------------------------------------------------
//-------------------------------------------------------------------------------------------------
int inIMAGetImaIndex(int *pinIndex, IMAGE *pIMIma)
{
  (*pinIndex) = pIMIma->inIndex;

  return(OK);
}

//-------------------------------------------------------------------------------------------------
//-------------------------------------------------------------------------------------------------
int inIMASetImaElem(IMAGE *pIMIma, int inRow, int inCol, mcomplex mcValue)
{
  int                          inStatus;

  inStatus = OK;

  pIMIma->ppmcElem[inRow][inCol].flRe = mcValue.flRe;
  pIMIma->ppmcElem[inRow][inCol].flIm = mcValue.flIm;

  return(inStatus);
}

//-------------------------------------------------------------------------------------------------
//-------------------------------------------------------------------------------------------------
int inIMAGetImaElem(mcomplex *pmcValue, IMAGE *pIMIma, int inRow, int inCol)
{
  int                          inStatus;

  inStatus = OK;

  pmcValue->flRe = pIMIma->ppmcElem[inRow][inCol].flRe;
  pmcValue->flIm = pIMIma->ppmcElem[inRow][inCol].flIm;
```

```
  return(inStatus);
}

//-------------------------------------------------------------------------------------------------
-------------------------
//-------------------------------------------------------------------------------------------------
-------------------------
IMAGE *pIMIMAReadIma(char *szName)
{
  int                          inHandle,
                                      inIndex,
                                      inNumRows,
                                      inNumCols,
                                      inCount1,
                                      inCount2;
  IMAGE                   *pIMIma;

  inHandle = _open(szName,_O_BINARY);

  if (inHandle < 0)
  {
    pIMIma = NULL;
  }
  else
  {
    read(inHandle,(void *)&inIndex,sizeof(int));
    read(inHandle,(void *)&inNumRows,sizeof(int));
    read(inHandle,(void *)&inNumCols,sizeof(int));

    pIMIma = pIMAskImaMem(inNumRows,inNumCols);

    if (pIMIma != NULL)
       {
      pIMIma->inIndex = inIndex;

      for (inCount1=0;inCount1<pIMIma->inNumRows;inCount1++)
         {
        for (inCount2=0;inCount2<pIMIma->inNumCols;inCount2++)
              {
          read(inHandle,(void *)&(pIMIma->ppmcElem[inCount1][inCount2].flRe),sizeof(float));
          read(inHandle,(void *)&(pIMIma->ppmcElem[inCount1][inCount2].flIm),sizeof(float));
              }
         }
       }

    _close(inHandle);
  }

  return(pIMIma);
}

//-------------------------------------------------------------------------------------------------
-------------------------
//-------------------------------------------------------------------------------------------------
-------------------------
int inIMASaveIma(char *szName, IMAGE *pIMIma)
{
  int                          inStatus,
                                      inHandle,
                                      inCount1,
                                      inCount2;

  inStatus = OK;

  inHandle = _open(szName,_O_CREAT|_O_WRONLY|_O_BINARY,_S_IREAD|_S_IWRITE);

  if (inHandle < 0)
  {
    inStatus = NK;
  }
  else
  {
    write(inHandle,(void *)&(pIMIma->inIndex),sizeof(int));
    write(inHandle,(void *)&(pIMIma->inNumRows),sizeof(int));
    write(inHandle,(void *)&(pIMIma->inNumCols),sizeof(int));
```

```
    for (inCount1=0;inCount1<pIMIma->inNumRows;inCount1++)
       {
      for (inCount2=0;inCount2<pIMIma->inNumCols;inCount2++)
         {
        write(inHandle,(void *)&(pIMIma->ppmcElem[inCount1][inCount2].flRe),sizeof(float));
        write(inHandle,(void *)&(pIMIma->ppmcElem[inCount1][inCount2].flIm),sizeof(float));
         }
       }

    _close(inHandle);
  }

  return(inStatus);
}

//-------------------------------------------------------------------------------------------------
-------------------------
//-------------------------------------------------------------------------------------------------
-------------------------
int inIMAImaMin(float *pflMin, IMAGE *pIMIma, int inType)
{
  return(inImaMin(pflMin,pIMIma,inType));
}

//-------------------------------------------------------------------------------------------------
-------------------------
//-------------------------------------------------------------------------------------------------
-------------------------
int inIMAImaMax(float *pflMax, IMAGE *pIMIma, int inType)
{
  return(inImaMax(pflMax,pIMIma,inType));
}

//-------------------------------------------------------------------------------------------------
-------------------------
//-------------------------------------------------------------------------------------------------
-------------------------
int inIMAImaMean(mcomplex *pmcMean, IMAGE *pIMIma)
{
  return(inImaMean(pmcMean,pIMIma));
}

//-------------------------------------------------------------------------------------------------
-------------------------
//-------------------------------------------------------------------------------------------------
-------------------------
int inIMAImaVar(mcomplex *pmcVar, IMAGE *pIMIma)
{
  return(inImaVar(pmcVar,pIMIma));
}

//-------------------------------------------------------------------------------------------------
-------------------------
//-------------------------------------------------------------------------------------------------
-------------------------
int inIMAInsertIma(IMAGE *pIMIma, IMAGE *pIMSub, int inRowIn, int inColIn)
{
  return(inInsertIma(pIMIma,pIMSub,inRowIn,inColIn));
}

//-------------------------------------------------------------------------------------------------
-------------------------
//-------------------------------------------------------------------------------------------------
-------------------------
int inIMAExtractIma(IMAGE *pIMSub, IMAGE *pIMIma, int inRowIn, int inColIn, int inDelRows, int
inDelCols)
{
  return(inExtractIma(pIMSub,pIMIma,inRowIn,inColIn,inDelRows,inDelCols));
}

//-------------------------------------------------------------------------------------------------
-------------------------
//-------------------------------------------------------------------------------------------------
-------------------------
int inIMACopyIma(IMAGE *pIMDesIma, IMAGE *pIMOriIma)
{
```

```
  return(inCopyIma(pIMDesIma,pIMOriIma));
}

//-------------------------------------------------------------------------------------------------
-------------------------
//-------------------------------------------------------------------------------------------------
-------------------------
int inIMAHorFlipIma(IMAGE *pIMOutIma, IMAGE *pIMInIma)
{
  return(inHorFlipIma(pIMOutIma,pIMInIma));
}

//-------------------------------------------------------------------------------------------------
-------------------------
//-------------------------------------------------------------------------------------------------
-------------------------
int inIMAVerFlipIma(IMAGE *pIMOutIma, IMAGE *pIMInIma)
{
  return(inVerFlipIma(pIMOutIma,pIMInIma));
}

//-------------------------------------------------------------------------------------------------
-------------------------
//-------------------------------------------------------------------------------------------------
-------------------------
int inIMAResample(IMAGE *pIMOutIma, IMAGE *pIMInIma)
{
  int                          inStatus,
                                      inFirSeg,
                                      inSecSeg;
  IMAGE                     *pIMBuffer1,
                                  *pIMBuffer2,
                                  *pIMBuffer3;

  inStatus = OK;

  if ((pIMOutIma->inNumRows != pIMInIma->inNumRows) && (pIMOutIma->inNumCols != pIMInIma-
>inNumCols))
  {
    // Gets the spectrum of the incoming image
    pIMBuffer1 = pIMIMAAskImaMem(pIMInIma->inNumRows,pIMInIma->inNumCols);
    inDFTIma(pIMBuffer1,pIMInIma,DFT);

    // Row resampling
    pIMBuffer2 = pIMIMAAskImaMem(pIMOutIma->inNumRows,pIMBuffer1->inNumCols);
    if (pIMBuffer2->inNumRows < pIMBuffer1->inNumRows)
       {
      inFirSeg = ((int)ceil(((float)pIMBuffer2->inNumRows)/2));
         inSecSeg = pIMBuffer2->inNumRows-inFirSeg;

      pIMBuffer3 = pIMAskImaMem(inFirSeg,pIMBuffer2->inNumCols);
         inExtractIma(pIMBuffer3,pIMBuffer1,0,0,inFirSeg,pIMBuffer2->inNumCols);
      inInsertIma(pIMBuffer2,pIMBuffer3,0,0);
      inFreeImaMem(pIMBuffer3);

      pIMBuffer3 = pIMAskImaMem(inSecSeg,pIMBuffer2->inNumCols);
         inExtractIma(pIMBuffer3,pIMBuffer1,pIMBuffer1->inNumRows-inSecSeg,0,inSecSeg,pIMBuffer2-
>inNumCols);
      inInsertIma(pIMBuffer2,pIMBuffer3,inFirSeg,0);
      inFreeImaMem(pIMBuffer3);
       }
    else
       {
      inZeroPadImage(pIMBuffer2,pIMBuffer1,SPREAD);
       }

    // frees the memory used to store the original spectrum
    inFreeImaMem(pIMBuffer1);

    // Column Resampling
    pIMBuffer1 = pIMIMAAskImaMem(pIMOutIma->inNumRows,pIMOutIma->inNumCols);
    if (pIMBuffer1->inNumCols < pIMBuffer2->inNumCols)
       {
      inFirSeg = ((int)ceil(((float)pIMBuffer1->inNumCols)/2));
         inSecSeg = pIMBuffer1->inNumCols-inFirSeg;
```

```
      pIMBuffer3 = pIMAskImaMem(pIMBuffer1->inNumRows,inFirSeg);
          inExtractIma(pIMBuffer3,pIMBuffer2,0,0,pIMBuffer1->inNumRows,inFirSeg);
      inInsertIma(pIMBuffer1,pIMBuffer3,0,0);
      inFreeImaMem(pIMBuffer3);

      pIMBuffer3 = pIMAskImaMem(pIMBuffer1->inNumRows,inSecSeg);
          inExtractIma(pIMBuffer3,pIMBuffer2,0,pIMBuffer2->inNumCols-inSecSeg,pIMBuffer1-
>inNumRows,inSecSeg);
      inInsertIma(pIMBuffer1,pIMBuffer3,0,inSecSeg);
      inFreeImaMem(pIMBuffer3);
        }
    else
        {
      inZeroPadImage(pIMBuffer1,pIMBuffer2,SPREAD);
        }

    // frees the memory used to store the intermediate spectrum
    inFreeImaMem(pIMBuffer2);

    inDFTIma(pIMOutIma,pIMBuffer1,IDFT);

    // frees the memory used to store the final spectrum
    inFreeImaMem(pIMBuffer1);
  }
  else
  {
    inCopyIma(pIMOutIma,pIMInIma);
  }

  return(inStatus);
}

//-------------------------------------------------------------------------------------------------
-------------------------
//-------------------------------------------------------------------------------------------------
-------------------------
int inIMAImaScaOp(IMAGE *pIMImaB, IMAGE *pIMImaA, mcomplex mcValue, int inOpType)
{
  return(inImaScaOp(pIMImaB,pIMImaA,mcValue,inOpType));
}

//-------------------------------------------------------------------------------------------------
-------------------------
//-------------------------------------------------------------------------------------------------
-------------------------
int inIMAImaMatOp(IMAGE *pIMImaC, IMAGE *pIMImaA, IMAGE *pIMImaB, int inOpType)
{
  return(inImaMatOp(pIMImaC,pIMImaA,pIMImaB,inOpType));
}

//-------------------------------------------------------------------------------------------------
-------------------------
//-------------------------------------------------------------------------------------------------
-------------------------
int inIMAImaFunOp(IMAGE *pIMImaB, IMAGE *pIMImaA, int inOpType)
{
  return(inImaFunOp(pIMImaB,pIMImaA,inOpType));
}

//-------------------------------------------------------------------------------------------------
-------------------------
//-------------------------------------------------------------------------------------------------
-------------------------
int inIMADFTIma(IMAGE *pIMOutIma, IMAGE *pIMInIma, int inTransformType)
{
  return(inDFTIma(pIMOutIma,pIMInIma,inTransformType));
}

//-------------------------------------------------------------------------------------------------
-------------------------
//-------------------------------------------------------------------------------------------------
-------------------------
int inIMACLS(IMAGE *pIMf, IMAGE *pIMg, IMAGE *pIMh, float flGamma, int inType)
{
  int                       inStatus,
                                inCount1,
```

```
                                        inCount2,
                                        inXCenter,
                                        inYCenter;
  float                           flMag1,
                                        flMag2,
                                        flBuffer;
  mcomplex                        mcValue1,
                                        mcValue2;
  IMAGE                        *pIMBuffer1,
                                     *pIMBuffer2,
                                     *pIMBuffer3;

  inStatus = OK;

  pIMBuffer1 = pIMAskImaMem(pIMf->inNumRows,pIMf->inNumCols);

  inDFTIma(pIMBuffer1,pIMBuffer1,DFT);

  pIMBuffer2 = pIMAskImaMem(pIMf->inNumRows,pIMf->inNumCols);

  voSetCom(&mcValue1,0,0);

  for (inCount1=0;inCount1<pIMf->inNumRows;inCount1++)
    for (inCount2=0;inCount2<pIMf->inNumCols;inCount2++)
      voAssignCom(&(pIMBuffer2->ppmcElem[inCount1][inCount2]),mcValue1);

  inXCenter = pIMf->inNumRows/2;
  inYCenter = pIMf->inNumCols/2;

  voSetCom(&mcValue1,+1.00,0);
  voSetCom(&mcValue2,-0.25,0);

  voAssignCom(&(pIMBuffer2->ppmcElem[inXCenter][inYCenter]),mcValue1);
  voAssignCom(&(pIMBuffer2->ppmcElem[inXCenter-1][inYCenter]),mcValue1);
  voAssignCom(&(pIMBuffer2->ppmcElem[inXCenter+1][inYCenter]),mcValue1);
  voAssignCom(&(pIMBuffer2->ppmcElem[inXCenter][inYCenter-1]),mcValue1);
  voAssignCom(&(pIMBuffer2->ppmcElem[inXCenter][inYCenter+1]),mcValue1);

  pIMBuffer3 = pIMAskImaMem(pIMf->inNumRows,pIMf->inNumCols);

  inDFTIma(pIMBuffer3,pIMBuffer2,DFT);

  pIMBuffer3 = pIMAskImaMem(pIMf->inNumRows,pIMf->inNumCols);

  for (inCount1=0;inCount1<pIMf->inNumRows;inCount1++)
  {
    for (inCount2=0;inCount2<pIMf->inNumCols;inCount2++)
       {
      voMagCom(&flMag1,pIMBuffer1->ppmcElem[inCount1][inCount2]);
      voMagCom(&flMag2,pIMBuffer3->ppmcElem[inCount1][inCount2]);
      flBuffer = flMag1*flMag1+flGamma*flMag2*flMag2;
         voSetCom(&mcValue1,flBuffer,0);
      voConjugateCom(&mcValue2,pIMBuffer1->ppmcElem[inCount1][inCount2]);
      voDivCom(&(pIMBuffer2->ppmcElem[inCount1][inCount2]),mcValue2,mcValue1);
       }
  }

  inDFTIma(pIMBuffer1,pIMBuffer2,IDFT);
  inImaConvolution(pIMBuffer2,pIMBuffer1,pIMg,inType);
  inIMAImaFunOp(pIMf,pIMBuffer2,IMRA);

  inFreeImaMem(pIMBuffer1);
  inFreeImaMem(pIMBuffer2);
  inFreeImaMem(pIMBuffer3);

  return(inStatus);
}

//-------------------------------------------------------------------------------------------------
-------------------------
//-------------------------------------------------------------------------------------------------
-------------------------
int inIMAMaxLik(IMAGE *pIMf, IMAGE *pIMg, IMAGE *pIMh, IMAGE *pIMPrior, int inNumIter,
                              int inType)
{
  int                             inStatus,
```

```
                                    inCount1;
  IMAGE                      *pIMBuffer1,
                                    *pIMBuffer2;

  inStatus = OK;

  pIMBuffer1 = pIMAskImaMem(pIMf->inNumRows,pIMf->inNumCols);
  pIMBuffer2 = pIMAskImaMem(pIMf->inNumRows,pIMf->inNumCols);

  inIMACopyIma(pIMf,pIMPrior);

  for (inCount1=0;inCount1<inNumIter;inCount1++)
  {
    inImaConvolution(pIMBuffer1,pIMf,pIMh,inType);
    inImaMatOp(pIMBuffer2,pIMg,pIMBuffer1,DIV);
    inImaConvolution(pIMBuffer1,pIMBuffer2,pIMh,inType);
        inCopyIma(pIMBuffer2,pIMf);
    inImaMatOp(pIMf,pIMBuffer2,pIMBuffer1,MUL);
        inCopyIma(pIMBuffer2,pIMf);
        inImaFunOp(pIMf,pIMBuffer2,IMRA);
  }

  inFreeImaMem(pIMBuffer1);
  inFreeImaMem(pIMBuffer2);

  return(inStatus);
}

//-------------------------------------------------------------------------------------------------
--------------------------
//-------------------------------------------------------------------------------------------------
--------------------------
int inIMAMAP(IMAGE *pIMf, IMAGE *pIMg, IMAGE *pIMh, IMAGE *pIMPrior, int inNumIter,
                        int inType)
{
  int                        inStatus,
                                    inCount1;
  mcomplex                   mcValue;
  IMAGE                      *pIMBuffer1,
                                    *pIMBuffer2;

  inStatus = OK;

  pIMBuffer1 = pIMAskImaMem(pIMf->inNumRows,pIMf->inNumCols);
  pIMBuffer2 = pIMAskImaMem(pIMf->inNumRows,pIMf->inNumCols);

  inIMACopyIma(pIMf,pIMPrior);

  for (inCount1=0;inCount1<inNumIter;inCount1++)
  {
    inImaConvolution(pIMBuffer1,pIMf,pIMh,inType);
    inImaMatOp(pIMBuffer2,pIMg,pIMBuffer1,DIV);
    voSetCom(&mcValue,1,0);
    inImaScaOp(pIMBuffer1,pIMBuffer2,mcValue,SUB);
// CACA
//      inImaCorrelation(pIMBuffer2,pIMBuffer1,pIMh,inType);
inImaConvolution(pIMBuffer2,pIMBuffer1,pIMh,inType);
    inImaFunOp(pIMBuffer1,pIMBuffer2,EXP);
    inCopyIma(pIMBuffer2,pIMf);
    inImaMatOp(pIMf,pIMBuffer1,pIMBuffer2,MUL);
        inCopyIma(pIMBuffer1,pIMf);
        inImaFunOp(pIMf,pIMBuffer1,IMRA);
  }

  inFreeImaMem(pIMBuffer1);
  inFreeImaMem(pIMBuffer2);

  return(inStatus);
}

//-------------------------------------------------------------------------------------------------
--------------------------
//-------------------------------------------------------------------------------------------------
--------------------------
int inIMAElipsoidalAperture(IMAGE *pIMAperture, int inAxis1, int inAxis2)
{
```

```
  int                           inStatus,
                                        inHalfRow,
                                        inHalfCol,
                                        inCount1,
                                        inCount2;
  float                         flXCen,
                                        flYCen,
                                        flRadius;

  inStatus = OK;

  inHalfRow = pIMAperture->inNumRows/2;
  inHalfCol = pIMAperture->inNumCols/2;

  for (inCount1=0;inCount1<pIMAperture->inNumRows;inCount1++)
  {
       for (inCount2=0;inCount2<pIMAperture->inNumCols;inCount2++)
       {
      flXCen = ((float)(inCount1-inHalfRow));
      flYCen = ((float)(inCount2-inHalfCol));
          flRadius =
(flXCen*flXCen)/(((float)inAxis1)*((float)inAxis1))+(flYCen*flYCen)/(((float)inAxis2)*((float)inA
xis2));
          if (flRadius < 1)
          {
            pIMAperture->ppmcElem[inCount1][inCount2].flRe = 1;
              pIMAperture->ppmcElem[inCount1][inCount2].flIm = 0;
          }
          else
          {
            pIMAperture->ppmcElem[inCount1][inCount2].flRe = 0;
            pIMAperture->ppmcElem[inCount1][inCount2].flIm = 0;
          }
        }
  }

  return(inStatus);
}

//-------------------------------------------------------------------------------------------------
-------------------------
//-------------------------------------------------------------------------------------------------
-------------------------
int inIMASlantedStripAperture(IMAGE *pIMAperture, int inLength, int inWidth, float flAngle)
{
  int                           inStatus,
                                        inRow2,
                                        inCol2,
                                        inLength2,
                                        inWidth2,
                                        inCount1,
                                        inCount2,
                                        inXCen,
                                        inYCen,
                                        inRadius;

  inStatus = OK;

  inRow2 = (int)(pIMAperture->inNumRows/2);
  inCol2 = (int)(pIMAperture->inNumCols/2);

  inLength2 = (int)(inLength/2);
  inWidth2 = (int)(inWidth/2);

  for (inCount1=0;inCount1<pIMAperture->inNumRows;inCount1++)
  {
       for (inCount2=0;inCount2<pIMAperture->inNumCols;inCount2++)
       {
      inXCen = inCount1-inRow2;
      inYCen = inCount2-inCol2;
          inRadius = (int)sqrt(inXCen*inXCen+inYCen*inYCen);
          if ((inXCen*cos(flAngle)+inYCen*sin(flAngle)-inLength2 < 0) &&
                  (inXCen*cos(PI+flAngle)+inYCen*sin(PI+flAngle)-inLength2 < 0) &&
                  (inXCen*cos(PI/2+flAngle)+inYCen*sin(PI/2+flAngle)-inWidth2 < 0) &&
                  (inXCen*cos(3*PI/2+flAngle)+inYCen*sin(3*PI/2+flAngle)-inWidth2 < 0))
          {
```

```
            pIMAperture->ppmcElem[inCount1][inCount2].flRe = 1;
                pIMAperture->ppmcElem[inCount1][inCount2].flIm = 0;
          }
          else
          {
            pIMAperture->ppmcElem[inCount1][inCount2].flRe = 0;
            pIMAperture->ppmcElem[inCount1][inCount2].flIm = 0;
          }
        }
  }

  return(inStatus);
}

//-------------------------------------------------------------------------------------------------
//-------------------------------------------------------------------------------------------------
int inIMAImaConvolution(IMAGE *pIMImaC, IMAGE *pIMImaA, IMAGE *pIMImaB, int inConType)
{
  return(inImaConvolution(pIMImaC,pIMImaA,pIMImaB,inConType));
}

//-------------------------------------------------------------------------------------------------
//-------------------------------------------------------------------------------------------------
int inIMAImaCorrelation(IMAGE *pIMImaC, IMAGE *pIMImaA, IMAGE *pIMImaB, int inCorType)
{
  return(inImaCorrelation(pIMImaC,pIMImaA,pIMImaB,inCorType));
}

//-------------------------------------------------------------------------------------------------
//-------------------------------------------------------------------------------------------------
int inIMAOTFFromAperture(IMAGE *pIMOTF, IMAGE *pIMAperture, int inConType)

{
  return(inOTFFromAperture(pIMOTF,pIMAperture,inConType));
}

//-------------------------------------------------------------------------------------------------
//-------------------------------------------------------------------------------------------------
int inIMAPSFFromAperture(IMAGE *pIMPSF, IMAGE *pIMAperture, int inConType)
{
  int                       inStatus;
  IMAGE                  *pIMBuffer1,
                            *pIMBuffer2;

  inStatus = OK;

  pIMBuffer1 = pIMAskImaMem(pIMAperture->inNumRows,pIMAperture->inNumCols);
  pIMBuffer2 = pIMAskImaMem(pIMAperture->inNumRows,pIMAperture->inNumCols);

  inOTFFromAperture(pIMBuffer1,pIMAperture,inConType);
  inDFTIma(pIMPSF,pIMBuffer1,IDFT);

  inFreeImaMem(pIMBuffer1);
  inFreeImaMem(pIMBuffer2);

  return(inStatus);
}

//-------------------------------------------------------------------------------------------------
//-------------------------------------------------------------------------------------------------
int inIMAStretch(IMAGE *pIMOutIma, IMAGE *pIMInIma, int inMinVal, int inMaxVal)
{
  int                       inStatus,
                               inCount1,
                               inCount2;
  float                     doMin,
```

```
                                        doMax,
                                        flBuffer1,
                                        flBuffer2;
  mcomplex                        mcValue1,
                                        mcValue2;

  inStatus = OK;

  inImaMin(&doMin,pIMInIma,REAL);
  inImaMax(&doMax,pIMInIma,REAL);

  if (doMin != doMax)
  {
    flBuffer1 = ((float)inMaxVal-(float)inMinVal)/(doMax-doMin);
       flBuffer2 = -flBuffer1*doMin+((float)inMinVal);

       for (inCount1=0;inCount1<pIMInIma->inNumRows;inCount1++)
       {
         for (inCount2=0;inCount2<pIMInIma->inNumCols;inCount2++)
         {
       inIMAGetImaElem(&mcValue1,pIMInIma,inCount1,inCount2);
       voSetCom(&mcValue2,flBuffer1*mcValue1.flRe+flBuffer2,flBuffer1*mcValue1.flIm+flBuffer2);
       inIMASetImaElem(pIMOutIma,inCount1,inCount2,mcValue2);
         }
       }
  }

  return(inStatus);
}

//-------------------------------------------------------------------------------------------------
-------------------------
//-------------------------------------------------------------------------------------------------
-------------------------
int inIMAImaMetric(float *pdoDistance, IMAGE *pIMA, IMAGE *pIMB, int inMetricType, float flParam)
{
  return(inImaMetric(pdoDistance,pIMA,pIMB,inMetricType,flParam));
}

//-------------------------------------------------------------------------------------------------
-------------------------
//-------------------------------------------------------------------------------------------------
-------------------------
int inIMAImaShift(IMAGE *pIMImaOut, IMAGE *pIMImaIn)
{
  return(inImaShift(pIMImaOut,pIMImaIn));
}

//-------------------------------------------------------------------------------------------------
-------------------------
//-------------------------------------------------------------------------------------------------
-------------------------
int inIMASimMap(IMAGE *pIMMap, IMAGE *pIMRef, IMAGE *pIMIma, int inBlockSize, int inMetricType,
float flParam)
{
  return(inSimMap(pIMMap,pIMRef,pIMIma,inBlockSize,inMetricType,flParam));
}

//-------------------------------------------------------------------------------------------------
-------------------------
//-------------------------------------------------------------------------------------------------
-------------------------
int inIMAAverageSimilarity(float *pflAveSim, int inLimit, IMAGE *pIMIdeal, IMAGE *pIMImage)
{
  int                       inStatus,
                                inRowCenter,
                                inColCenter,
                                inCount1,
                                inCount2,
                                inCount3;
  float                     flRadius,
                                flBuffer1,
                                flMagnitude,
                                flEnergy;
  IMAGE                  *pIMBuffer1,
                             *pIMBuffer2,
```

```
                                    *pIMBuffer3;

  inStatus = OK;

  pIMBuffer1 = pIMAskImaMem(pIMIdeal->inNumRows,pIMIdeal->inNumCols);
  pIMBuffer2 = pIMAskImaMem(pIMIdeal->inNumRows,pIMIdeal->inNumCols);
  pIMBuffer3 = pIMAskImaMem(pIMIdeal->inNumRows,pIMIdeal->inNumCols);

  inDFTIma(pIMBuffer1,pIMIdeal,DFT);
  inImaShift(pIMBuffer2,pIMBuffer1);

  inDFTIma(pIMBuffer1,pIMImage,DFT);
  inImaShift(pIMBuffer3,pIMBuffer1);

  inSimMap(pIMBuffer1,pIMBuffer2,pIMBuffer3,4,CORRELATION,0);

  inRowCenter = (int)(floor(pIMIdeal->inNumRows/2));
  inColCenter = (int)(floor(pIMIdeal->inNumCols/2));

  for (inCount1=0;inCount1<inLimit;inCount1++)
  {
    pflAveSim[inCount1] = 0;
        flBuffer1 = 0;
    for (inCount2=0;inCount2<pIMIdeal->inNumRows;inCount2++)
        {
      for (inCount3=0;inCount3<pIMIdeal->inNumCols;inCount3++)
          {
        flRadius = (float)sqrt((inCount2-inRowCenter)*(inCount2-inRowCenter)+(inCount3-
inColCenter)*(inCount3-inColCenter));
             if ((flRadius < inCount1+5) && (inCount1-5 < flRadius))
             {
          voMagCom(&flMagnitude,pIMBuffer2->ppmcElem[inCount2][inCount3]);
              flEnergy = flMagnitude*flMagnitude;
          pflAveSim[inCount1] += (flEnergy*pIMBuffer1->ppmcElem[inCount2][inCount3].flRe);
              flBuffer1 += flEnergy;
             }
          }
        }
        pflAveSim[inCount1] /= flBuffer1;
  }

  inFreeImaMem(pIMBuffer1);
  inFreeImaMem(pIMBuffer2);
  inFreeImaMem(pIMBuffer3);

  return(inStatus);
}

//-------------------------------------------------------------------------------------------------
-------------------------
//-------------------------------------------------------------------------------------------------
-------------------------
int inIMAZeroPadImage(IMAGE *pIMOutIma, IMAGE *pIMInIma, int inType)
{
  return(inZeroPadImage(pIMOutIma,pIMInIma,inType));
}

//-------------------------------------------------------------------------------------------------
-------------------------
//-------------------------------------------------------------------------------------------------
-------------------------
int inIMAGetWindow(IMAGE *pIMWindow, int inWindowType)
{
  return(inGetWindow(pIMWindow,inWindowType));
}

//-------------------------------------------------------------------------------------------------
-------------------------
//-------------------------------------------------------------------------------------------------
-------------------------
int inIMALikelihood(float *pflLik, IMAGE *pIMg, IMAGE *pIMEstBlur, IMAGE *pIMf,
                                   IMAGE *pIMPrior, int inType)
{
  return(inLikelihood(pflLik,pIMg,pIMEstBlur,pIMf,pIMPrior,inType));
}
```

```
//-------------------------------------------------------------------------------------------------
-------------------------
//-------------------------------------------------------------------------------------------------
-------------------------
int inIMATotalFlip(IMAGE *pIMOutIma, IMAGE *pIMInIma)
{
  return(inTotalFlip(pIMOutIma,pIMInIma));
}

//-------------------------------------------------------------------------------------------------
-------------------------
// PRIVATE ROUTINES
//-------------------------------------------------------------------------------------------------
-------------------------

//-------------------------------------------------------------------------------------------------
-------------------------
//-------------------------------------------------------------------------------------------------
-------------------------
IMAGE *pIMAskImaMem(int inNumRows, int inNumCols)
{
  int                          inStatus,
                                      inCount1;
  IMAGE                    *pIMIma;

  inStatus = OK;

  pIMIma = (IMAGE *) calloc(1,sizeof(IMAGE));

  if (pIMIma == NULL)
  {
    inStatus = NK;
  }
  else
  {
    pIMIma->inIndex = 0;

    pIMIma->inNumRows = inNumRows;
    pIMIma->inNumCols = inNumCols;

    pIMIma->ppmcElem = (mcomplex **) calloc(inNumRows,sizeof(mcomplex *));
        if (pIMIma->ppmcElem == NULL)
        {
      inStatus = NK;
        }
        else
        {
      for (inCount1=0;inCount1<inNumRows;inCount1++)
          {
         pIMIma->ppmcElem[inCount1] = (mcomplex *) calloc(inNumCols,sizeof(mcomplex));
             if (pIMIma->ppmcElem[inCount1] == NULL)
                {
           inStatus = NK;
                }
          }
        }
  }

  if (inStatus == NK)
    inFreeImaMem(pIMIma);

  return(pIMIma);
}

//-------------------------------------------------------------------------------------------------
-------------------------
//-------------------------------------------------------------------------------------------------
-------------------------
int inFreeImaMem(IMAGE *pIMIma)
{
  int                          inStatus,
                                      inCount1;

  inStatus = OK;

  if (pIMIma != NULL)
```

```
  {
       if (pIMIma->ppmcElem != NULL)
    {
      for (inCount1=0;inCount1<pIMIma->inNumRows;inCount1++)
           if (pIMIma->ppmcElem[inCount1] != NULL)
                free(pIMIma->ppmcElem[inCount1]);

      free(pIMIma->ppmcElem);
       }

    free(pIMIma);
  }

  return(inStatus);
}

//-------------------------------------------------------------------------------------------------------
-------------------------
//-------------------------------------------------------------------------------------------------------
-------------------------
int inImaMin(float *pflMin, IMAGE *pIMIma, int inType)
{
  int                          inStatus,
                                      inCount1,
                                      inCount2;
  float                        flBuffer;

  inStatus = OK;

  switch (inType)
  {
    case REAL:
      (*pflMin) = pIMIma->ppmcElem[0][0].flRe;
      for (inCount1=0;inCount1<pIMIma->inNumRows;inCount1++)
         {
        for (inCount2=0;inCount2<pIMIma->inNumCols;inCount2++)
             {
          flBuffer = pIMIma->ppmcElem[inCount1][inCount2].flRe;
              if (flBuffer < (*pflMin))
                (*pflMin) = flBuffer;
             }
        }
        break;

    case IMAGINARY:
      (*pflMin) = pIMIma->ppmcElem[0][0].flIm;
      for (inCount1=0;inCount1<pIMIma->inNumRows;inCount1++)
         {
        for (inCount2=0;inCount2<pIMIma->inNumCols;inCount2++)
             {
          flBuffer = pIMIma->ppmcElem[inCount1][inCount2].flIm;
              if (flBuffer < (*pflMin))
                (*pflMin) = flBuffer;
             }
        }
        break;

    case MAGNITUDE:
      voMagCom(pflMin,pIMIma->ppmcElem[0][0]);
      for (inCount1=0;inCount1<pIMIma->inNumRows;inCount1++)
         {
        for (inCount2=0;inCount2<pIMIma->inNumCols;inCount2++)
             {
          voMagCom(&flBuffer,pIMIma->ppmcElem[inCount1][inCount2]);
              if (flBuffer < (*pflMin))
                (*pflMin) = flBuffer;
             }
        }
        break;
  }

  return(inStatus);
}

//-------------------------------------------------------------------------------------------------------
-------------------------
```

```
//-------------------------------------------------------------------------------------------------
-------------------------
int inImaMax(float *pflMax, IMAGE *pIMIma, int inType)
{
  int                           inStatus,
                                       inCount1,
                                       inCount2;
  float                         flBuffer;

  inStatus = OK;

  switch (inType)
  {
    case REAL:
      (*pflMax) = pIMIma->ppmcElem[0][0].flRe;
      for (inCount1=0;inCount1<pIMIma->inNumRows;inCount1++)
         {
        for (inCount2=0;inCount2<pIMIma->inNumCols;inCount2++)
               {
          flBuffer = pIMIma->ppmcElem[inCount1][inCount2].flRe;
              if (flBuffer > (*pflMax))
                (*pflMax) = flBuffer;
               }
         }
         break;

    case IMAGINARY:
      (*pflMax) = pIMIma->ppmcElem[0][0].flIm;
      for (inCount1=0;inCount1<pIMIma->inNumRows;inCount1++)
         {
        for (inCount2=0;inCount2<pIMIma->inNumCols;inCount2++)
               {
          flBuffer = pIMIma->ppmcElem[inCount1][inCount2].flIm;
              if (flBuffer > (*pflMax))
                (*pflMax) = flBuffer;
               }
         }
         break;

    case MAGNITUDE:
      voMagCom(pflMax,pIMIma->ppmcElem[0][0]);
      for (inCount1=0;inCount1<pIMIma->inNumRows;inCount1++)
         {
        for (inCount2=0;inCount2<pIMIma->inNumCols;inCount2++)
               {
          voMagCom(&flBuffer,pIMIma->ppmcElem[inCount1][inCount2]);
              if (flBuffer > (*pflMax))
                (*pflMax) = flBuffer;
               }
         }
         break;

    case PHASE:
      voPhaCom(pflMax,pIMIma->ppmcElem[0][0]);
      for (inCount1=0;inCount1<pIMIma->inNumRows;inCount1++)
         {
        for (inCount2=0;inCount2<pIMIma->inNumCols;inCount2++)
               {
          voPhaCom(&flBuffer,pIMIma->ppmcElem[inCount1][inCount2]);
              if (flBuffer > (*pflMax))
                (*pflMax) = flBuffer;
               }
         }
         break;
  }

  return(inStatus);
}

//-------------------------------------------------------------------------------------------------
-------------------------
//-------------------------------------------------------------------------------------------------
-------------------------
int inImaMean(mcomplex *pmcMean, IMAGE *pIMIma)
{
  int                           inStatus,
```

```
                                    inCount1,
                                    inCount2;
  mcomplex                  mcBuffer1,
                                    mcBuffer2;

  inStatus = OK;

  voSetCom(pmcMean,0,0);
  for (inCount1=0;inCount1<pIMIma->inNumRows;inCount1++)
  {
    for (inCount2=0;inCount2<pIMIma->inNumCols;inCount2++)
       {
      voAssignCom(&mcBuffer1,*pmcMean);
      voAddCom(pmcMean,pIMIma->ppmcElem[inCount1][inCount2],mcBuffer1);
        }
  }

  voAssignCom(&mcBuffer1,*pmcMean);
  voSetCom(&mcBuffer2,(float)(pIMIma->inNumRows*pIMIma->inNumCols),0);
  voDivCom(pmcMean,mcBuffer1,mcBuffer2);

  return(inStatus);
}

//-------------------------------------------------------------------------------------------------
--------------------------
//-------------------------------------------------------------------------------------------------
--------------------------
int inImaVar(mcomplex *pmcVar, IMAGE *pIMIma)
{
  int                       inStatus,
                                    inCount1,
                                    inCount2;
  mcomplex                  mcMean,
                                    mcBuffer1,
                                    mcBuffer2;

  inStatus = OK;

  inImaMean(&mcMean,pIMIma);

  voSetCom(pmcVar,0,0);
  for (inCount1=0;inCount1<pIMIma->inNumRows;inCount1++)
  {
    for (inCount2=0;inCount2<pIMIma->inNumCols;inCount2++)
       {
      voConjugateCom(&mcBuffer1,pIMIma->ppmcElem[inCount1][inCount2]);
      voMulCom(&mcBuffer2,pIMIma->ppmcElem[inCount1][inCount2],mcBuffer1);
      voAssignCom(&mcBuffer1,*pmcVar);
      voAddCom(pmcVar,mcBuffer1,mcBuffer2);
        }
  }

  voAssignCom(&mcBuffer1,*pmcVar);
  voSetCom(&mcBuffer2,(float)(pIMIma->inNumRows*pIMIma->inNumCols),0);
  voDivCom(pmcVar,mcBuffer1,mcBuffer2);

  voConjugateCom(&mcBuffer1,mcMean);
  voMulCom(&mcBuffer2,mcMean,mcBuffer1);
  voAssignCom(&mcBuffer1,*pmcVar);
  voSubCom(pmcVar,mcBuffer1,mcBuffer2);

  return(inStatus);
}

//-------------------------------------------------------------------------------------------------
--------------------------
//-------------------------------------------------------------------------------------------------
--------------------------
int inIsWithinScope(IMAGE *pIMIma, int inRow, int inCol)
{
  int                       inResult;

  inResult = YES;

  if ((inRow < 0) || (pIMIma->inNumRows <= inRow))
```

```
    inResult = NO;

  if ((inCol < 0) || (pIMIma->inNumCols <= inCol))
    inResult = NO;

  return(inResult);
}

//-------------------------------------------------------------------------------------------------
-------------------------
//-------------------------------------------------------------------------------------------------
-------------------------
int inInsertIma(IMAGE *pIMIma, IMAGE *pIMSub, int inRowIn, int inColIn)
{
  int                           inStatus,
                                inRowLim,
                                inColLim,
                                inCount1,
                                inCount2;

  inStatus = OK;

  inRowLim = inRowIn+pIMSub->inNumRows;
  inColLim = inColIn+pIMSub->inNumCols;

  for (inCount1=inRowIn;inCount1<inRowLim;inCount1++)
  {
    for (inCount2=inColIn;inCount2<inColLim;inCount2++)
       {
      if (inIsWithinScope(pIMIma,inCount1,inCount2) == YES)
         {
        if (inIsWithinScope(pIMSub,inCount1-inRowIn,inCount2-inColIn) == YES)
          voAssignCom(&(pIMIma->ppmcElem[inCount1][inCount2]),pIMSub->ppmcElem[inCount1-
inRowIn][inCount2-inColIn]);
              else
          voSetCom(&(pIMIma->ppmcElem[inCount1][inCount2]),0,0);
         }
       }
  }

  return(inStatus);
}

//-------------------------------------------------------------------------------------------------
-------------------------
//-------------------------------------------------------------------------------------------------
-------------------------
int inExtractIma(IMAGE *pIMSub, IMAGE *pIMIma, int inRowIn, int inColIn, int inDelRows, int
inDelCols)
{
  int                           inStatus,
                                inRowLim,
                                inColLim,
                                inCount1,
                                inCount2;

  inStatus = OK;

  inRowLim = inRowIn+inDelRows;
  inColLim = inColIn+inDelCols;

  for (inCount1=inRowIn;inCount1<inRowLim;inCount1++)
  {
    for (inCount2=inColIn;inCount2<inColLim;inCount2++)
       {
      if (inIsWithinScope(pIMSub,inCount1-inRowIn,inCount2-inColIn) == YES)
         {
        if (inIsWithinScope(pIMIma,inCount1,inCount2) == YES)
          voAssignCom(&(pIMSub->ppmcElem[inCount1-inRowIn][inCount2-inColIn]),pIMIma-
>ppmcElem[inCount1][inCount2]);
            else
          voSetCom(&(pIMSub->ppmcElem[inCount1-inRowIn][inCount2-inColIn]),0,0);
         }
       }
  }
```

```
  return(inStatus);
}

//-------------------------------------------------------------------------------------------------
--------------------------
//-------------------------------------------------------------------------------------------------
--------------------------
int inCopyIma(IMAGE *pIMDesIma, IMAGE *pIMOriIma)
{
  return(inExtractIma(pIMDesIma,pIMOriIma,0,0,pIMOriIma->inNumRows,pIMOriIma->inNumCols));
}

//-------------------------------------------------------------------------------------------------
--------------------------
//-------------------------------------------------------------------------------------------------
--------------------------
int inHorFlipIma(IMAGE *pIMOutIma, IMAGE *pIMInIma)
{
  int                           inStatus,
                                inCount1,
                                inCount2;

  inStatus = OK;

  for (inCount1=0;inCount1<pIMInIma->inNumRows;inCount1++)
    for (inCount2=0;inCount2<pIMInIma->inNumCols;inCount2++)
        voAssignCom(&(pIMOutIma->ppmcElem[pIMInIma->inNumRows-inCount1-1][inCount2]),pIMInIma-
>ppmcElem[inCount1][inCount2]);

  return(inStatus);
}

//-------------------------------------------------------------------------------------------------
--------------------------
//-------------------------------------------------------------------------------------------------
--------------------------
int inVerFlipIma(IMAGE *pIMOutIma, IMAGE *pIMInIma)
{
  int                           inStatus,
                                inCount1,
                                inCount2;

  inStatus = OK;

  for (inCount1=0;inCount1<pIMInIma->inNumRows;inCount1++)
    for (inCount2=0;inCount2<pIMInIma->inNumCols;inCount2++)
        voAssignCom(&(pIMOutIma->ppmcElem[inCount1][pIMInIma->inNumCols-inCount2-1]),pIMInIma-
>ppmcElem[inCount1][inCount2]);

  return(inStatus);
}

//-------------------------------------------------------------------------------------------------
--------------------------
//-------------------------------------------------------------------------------------------------
--------------------------
int inImaFunOp(IMAGE *pIMImaB, IMAGE *pIMImaA, int inOpType)
{
  int                           inStatus,
                                inCount1,
                                inCount2;
  float                         flBuffer1;

  inStatus = OK;

  switch (inOpType)
  {
      case ZERE:
    for (inCount1=0;inCount1<pIMImaB->inNumRows;inCount1++)
      for (inCount2=0;inCount2<pIMImaB->inNumCols;inCount2++)
            voSetCom(&(pIMImaB->ppmcElem[inCount1][inCount2]),0,pIMImaA-
>ppmcElem[inCount1][inCount2].flIm);
    break;

      case ZEIM:
    for (inCount1=0;inCount1<pIMImaB->inNumRows;inCount1++)
```

```
        for (inCount2=0;inCount2<pIMImaB->inNumCols;inCount2++)
                voSetCom(&(pIMImaB->ppmcElem[inCount1][inCount2]),pIMImaA-
>ppmcElem[inCount1][inCount2].flRe,0);
      break;

    case IMRA:
      for (inCount1=0;inCount1<pIMImaB->inNumRows;inCount1++)
        for (inCount2=0;inCount2<pIMImaB->inNumCols;inCount2++)
          if (pIMImaA->ppmcElem[inCount1][inCount2].flRe < 0)
                  voSetCom(&(pIMImaB->ppmcElem[inCount1][inCount2]),0,0);
                else
                  voSetCom(&(pIMImaB->ppmcElem[inCount1][inCount2]),pIMImaA-
>ppmcElem[inCount1][inCount2].flRe,0);
      break;

       case EXP:
      for (inCount1=0;inCount1<pIMImaB->inNumRows;inCount1++)
        for (inCount2=0;inCount2<pIMImaB->inNumCols;inCount2++)
                voExpCom(&pIMImaB->ppmcElem[inCount1][inCount2],pIMImaA-
>ppmcElem[inCount1][inCount2]);
      break;

       case LOG:
      for (inCount1=0;inCount1<pIMImaB->inNumRows;inCount1++)
        for (inCount2=0;inCount2<pIMImaB->inNumCols;inCount2++)
                voLogCom(&pIMImaB->ppmcElem[inCount1][inCount2],pIMImaA-
>ppmcElem[inCount1][inCount2]);
      break;

       case MAGNITUDE:
      for (inCount1=0;inCount1<pIMImaB->inNumRows;inCount1++)
         {
        for (inCount2=0;inCount2<pIMImaB->inNumCols;inCount2++)
              {
                voMagCom((float *)&flBuffer1,pIMImaA->ppmcElem[inCount1][inCount2]);
                voSetCom(&(pIMImaB->ppmcElem[inCount1][inCount2]),flBuffer1,0);
              }
         }
      break;

       case PHASE:
      for (inCount1=0;inCount1<pIMImaB->inNumRows;inCount1++)
         {
        for (inCount2=0;inCount2<pIMImaB->inNumCols;inCount2++)
              {
                voPhaCom((float *)&flBuffer1,pIMImaA->ppmcElem[inCount1][inCount2]);
                voSetCom(&(pIMImaB->ppmcElem[inCount1][inCount2]),flBuffer1,0);
              }
         }
      break;

       default:
         inStatus = NK;
         break;
  }

  return(inStatus);
}

//-------------------------------------------------------------------------------------------------
--------------------------
//-------------------------------------------------------------------------------------------------
--------------------------
int inImaScaOp(IMAGE *pIMImaB, IMAGE *pIMImaA, mcomplex mcValue, int inOpType)
{
  int                          inStatus,
                                inCount1,
                                inCount2;

  inStatus = OK;

  switch (inOpType)
  {
       case ADD:
      for (inCount1=0;inCount1<pIMImaB->inNumRows;inCount1++)
        for (inCount2=0;inCount2<pIMImaB->inNumCols;inCount2++)
```

```
                voAddCom(&pIMImaB->ppmcElem[inCount1][inCount2],pIMImaA-
>ppmcElem[inCount1][inCount2],mcValue);
      break;

       case SUB:
      for (inCount1=0;inCount1<pIMImaB->inNumRows;inCount1++)
        for (inCount2=0;inCount2<pIMImaB->inNumCols;inCount2++)
             voSubCom(&pIMImaB->ppmcElem[inCount1][inCount2],pIMImaA-
>ppmcElem[inCount1][inCount2],mcValue);
      break;

       case MUL:
      for (inCount1=0;inCount1<pIMImaB->inNumRows;inCount1++)
        for (inCount2=0;inCount2<pIMImaB->inNumCols;inCount2++)
                voMulCom(&pIMImaB->ppmcElem[inCount1][inCount2],pIMImaA-
>ppmcElem[inCount1][inCount2],mcValue);
      break;

       case DIV:
      for (inCount1=0;inCount1<pIMImaB->inNumRows;inCount1++)
        for (inCount2=0;inCount2<pIMImaB->inNumCols;inCount2++)
                voDivCom(&pIMImaB->ppmcElem[inCount1][inCount2],pIMImaA-
>ppmcElem[inCount1][inCount2],mcValue);
      break;

       case POW:
      for (inCount1=0;inCount1<pIMImaB->inNumRows;inCount1++)
        for (inCount2=0;inCount2<pIMImaB->inNumCols;inCount2++)
          voPowCom(&pIMImaB->ppmcElem[inCount1][inCount2],pIMImaA-
>ppmcElem[inCount1][inCount2],mcValue);
      break;

       default:
         inStatus = NK;
         break;
  }

  return(inStatus);
}

//-------------------------------------------------------------------------------------------------------
-------------------------
//-------------------------------------------------------------------------------------------------------
-------------------------
int inImaMatOp(IMAGE *pIMImaC, IMAGE *pIMImaA, IMAGE *pIMImaB, int inOpType)
{
  int                          inStatus,
                                      inCount1,
                                      inCount2;

  inStatus = OK;

  switch (inOpType)
  {
       case ADD:
      for (inCount1=0;inCount1<pIMImaC->inNumRows;inCount1++)
        for (inCount2=0;inCount2<pIMImaC->inNumCols;inCount2++)
             voAddCom(&pIMImaC->ppmcElem[inCount1][inCount2],
                              pIMImaA->ppmcElem[inCount1][inCount2],
                                 pIMImaB->ppmcElem[inCount1][inCount2]);
      break;

       case SUB:
      for (inCount1=0;inCount1<pIMImaC->inNumRows;inCount1++)
        for (inCount2=0;inCount2<pIMImaC->inNumCols;inCount2++)
                voSubCom(&pIMImaC->ppmcElem[inCount1][inCount2],
                                    pIMImaA->ppmcElem[inCount1][inCount2],
                                 pIMImaB->ppmcElem[inCount1][inCount2]);
      break;

       case MUL:
      for (inCount1=0;inCount1<pIMImaC->inNumRows;inCount1++)
        for (inCount2=0;inCount2<pIMImaC->inNumCols;inCount2++)
                voMulCom(&pIMImaC->ppmcElem[inCount1][inCount2],
                                    pIMImaA->ppmcElem[inCount1][inCount2],
                                 pIMImaB->ppmcElem[inCount1][inCount2]);
```

```
      break;

       case DIV:
      for (inCount1=0;inCount1<pIMImaC->inNumRows;inCount1++)
        for (inCount2=0;inCount2<pIMImaC->inNumCols;inCount2++)
                voDivCom(&pIMImaC->ppmcElem[inCount1][inCount2],
                                pIMImaA->ppmcElem[inCount1][inCount2],
                              pIMImaB->ppmcElem[inCount1][inCount2]);
      break;

       case POW:
      for (inCount1=0;inCount1<pIMImaC->inNumRows;inCount1++)
        for (inCount2=0;inCount2<pIMImaC->inNumCols;inCount2++)
          voPowCom(&pIMImaC->ppmcElem[inCount1][inCount2],
                              pIMImaA->ppmcElem[inCount1][inCount2],
                              pIMImaB->ppmcElem[inCount1][inCount2]);
      break;

       default:
         inStatus = NK;
         break;
  }

  return(inStatus);
}

//-------------------------------------------------------------------------------------------------------
-------------------------
//-------------------------------------------------------------------------------------------------------
-------------------------
int inDFTIma(IMAGE *pIMOutIma, IMAGE *pIMInIma, int inTransformType)
{
  int                           inStatus,
                                        inNumElem,
                                        inCount1,
                                        inCount2,
                                        inBase1,
                                        inBase2;

  unsigned long     *pulDims;
  float                     *pflData;

  inStatus = OK;

  inNumElem = (2*pIMInIma->inNumRows*pIMInIma->inNumCols);
  pflData = (float *) calloc(inNumElem,sizeof(float));

  for (inCount1=0;inCount1<pIMInIma->inNumRows;inCount1++)
  {
    inBase1 = inCount1*(2*pIMInIma->inNumCols);
    for (inCount2=0;inCount2<pIMInIma->inNumCols;inCount2++)
       {
      inBase2 = inBase1+(2*inCount2);
         pflData[inBase2] = pIMInIma->ppmcElem[inCount1][inCount2].flRe;
         pflData[inBase2+1] = pIMInIma->ppmcElem[inCount1][inCount2].flIm;
       }
  }

  pulDims = (unsigned long *) calloc(2,sizeof(unsigned long));

  pulDims[0] = pIMInIma->inNumRows;
  pulDims[1] = pIMInIma->inNumCols;

  voDFTNDFT(pflData,pulDims,2,inTransformType);

  for (inCount1=0;inCount1<pIMOutIma->inNumRows;inCount1++)
  {
    inBase1 = inCount1*(2*pIMOutIma->inNumCols);
    for (inCount2=0;inCount2<pIMOutIma->inNumCols;inCount2++)
       {
      inBase2 = inBase1+(2*inCount2);
      voSetCom(&(pIMOutIma->ppmcElem[inCount1][inCount2]),pflData[inBase2],pflData[inBase2+1]);
       }
  }

  free(pflData);
```

```
  free(pulDims);

  return(inStatus);
}

//-------------------------------------------------------------------------------------------------
-------------------------
//-------------------------------------------------------------------------------------------------
-------------------------
int inZeroPadImage(IMAGE *pIMOutIma, IMAGE *pIMInIma, int inType)
{
  int                          inStatus,
                                      inCount1,
                                      inCount2,
                                      inRowPos,
                                      inColPos,
                                      inRowFirSeg,
                                      inRowSecSeg,
                                      inColFirSeg,
                                      inColSecSeg;
  IMAGE            *pIMIma1;

  inStatus = OK;

  for (inCount1=0;inCount1<pIMOutIma->inNumRows;inCount1++)
    for (inCount2=0;inCount2<pIMOutIma->inNumCols;inCount2++)
      voSetCom(&(pIMOutIma->ppmcElem[inCount1][inCount2]),0,0);

  switch (inType)
  {
    case CORNERED:
      inInsertIma(pIMOutIma,pIMInIma,0,0);
          break;

    case CENTERED:
      inRowPos = ((int)floor(((float)(pIMOutIma->inNumRows-pIMInIma->inNumRows))/2));
      inColPos = ((int)floor(((float)(pIMOutIma->inNumCols-pIMInIma->inNumCols))/2));
      inInsertIma(pIMOutIma,pIMInIma,inRowPos,inColPos);
          break;

    case SPREAD:
      inRowFirSeg = ((int)ceil(((float)pIMInIma->inNumRows)/2));
          inRowSecSeg = pIMInIma->inNumRows-inRowFirSeg;
      inColFirSeg = ((int)ceil(((float)pIMInIma->inNumCols)/2));
          inColSecSeg = pIMInIma->inNumCols-inColFirSeg;

      pIMIma1 = pIMAskImaMem(inRowFirSeg,inColFirSeg);
          inExtractIma(pIMIma1,pIMInIma,0,0,inRowFirSeg,inColFirSeg);
      inInsertIma(pIMOutIma,pIMIma1,0,0);
      inFreeImaMem(pIMIma1);

      pIMIma1 = pIMAskImaMem(inRowSecSeg,inColFirSeg);
          inExtractIma(pIMIma1,pIMInIma,inRowFirSeg,0,inRowSecSeg,inColFirSeg);
      inInsertIma(pIMOutIma,pIMIma1,pIMOutIma->inNumRows-inRowSecSeg,0);
      inFreeImaMem(pIMIma1);

      pIMIma1 = pIMAskImaMem(inRowFirSeg,inColSecSeg);
          inExtractIma(pIMIma1,pIMInIma,0,inColFirSeg,inRowFirSeg,inColSecSeg);
      inInsertIma(pIMOutIma,pIMIma1,0,pIMOutIma->inNumCols-inColSecSeg);
      inFreeImaMem(pIMIma1);

      pIMIma1 = pIMAskImaMem(inRowSecSeg,inColSecSeg);
          inExtractIma(pIMIma1,pIMInIma,inRowFirSeg,inColFirSeg,inRowSecSeg,inColSecSeg);
      inInsertIma(pIMOutIma,pIMIma1,pIMOutIma->inNumRows-inRowSecSeg,pIMOutIma->inNumCols-
inColSecSeg);
      inFreeImaMem(pIMIma1);

      break;
  }

  return(inStatus);
}

//-------------------------------------------------------------------------------------------------
-------------------------
```

```
//-------------------------------------------------------------------------------------------------
-------------------------
int inImaConvolution(IMAGE *pIMImaC, IMAGE *pIMImaA, IMAGE *pIMImaB, int inConType)
{
  int                           inStatus,
                                       inTotRow,
                                       inTotCol,
                                       inExtRow,
                                       inExtCol;
  IMAGE                     *pIMBuffer1,
                                   *pIMBuffer2,
                                   *pIMBuffer3;

  inStatus = OK;

  if (inConType == LINEAR)
  {
    inTotRow = pIMImaA->inNumRows+pIMImaB->inNumRows-1;
    inTotCol = pIMImaA->inNumCols+pIMImaB->inNumCols-1;

    pIMBuffer1 = pIMAskImaMem(inTotRow,inTotCol);
    pIMBuffer2 = pIMAskImaMem(inTotRow,inTotCol);
    pIMBuffer3 = pIMAskImaMem(inTotRow,inTotCol);

    inZeroPadImage(pIMBuffer1,pIMImaA,CORNERED);
    inZeroPadImage(pIMBuffer2,pIMImaB,CORNERED);

    inDFTIma(pIMBuffer3,pIMBuffer1,DFT);
    inIMACopyIma(pIMBuffer1,pIMBuffer3);
    inDFTIma(pIMBuffer3,pIMBuffer2,DFT);
    inIMACopyIma(pIMBuffer2,pIMBuffer3);
    inImaMatOp(pIMBuffer3,pIMBuffer1,pIMBuffer2,MUL);
    inDFTIma(pIMBuffer1,pIMBuffer3,IDFT);

    inExtRow = ((int)floor(((float)(pIMBuffer1->inNumRows-pIMImaC->inNumRows))/2));
    inExtCol = ((int)floor(((float)(pIMBuffer1->inNumCols-pIMImaC->inNumCols))/2));

    inExtractIma(pIMBuffer2,pIMBuffer1,inExtRow,inExtCol,pIMImaC->inNumRows,pIMImaC->inNumCols);
    inImaShift(pIMImaC,pIMBuffer2);

    inFreeImaMem(pIMBuffer1);
    inFreeImaMem(pIMBuffer2);
    inFreeImaMem(pIMBuffer3);
  }
  else if (inConType == CIRCULAR)
  {
    pIMBuffer1 = pIMAskImaMem(pIMImaA->inNumRows,pIMImaA->inNumCols);
    pIMBuffer2 = pIMAskImaMem(pIMImaB->inNumRows,pIMImaB->inNumCols);
    pIMBuffer3 = pIMAskImaMem(pIMImaC->inNumRows,pIMImaC->inNumCols);

        inDFTIma(pIMBuffer1,pIMImaA,DFT);
    inDFTIma(pIMBuffer2,pIMImaB,DFT);
    inImaMatOp(pIMBuffer3,pIMBuffer1,pIMBuffer2,MUL);
    inDFTIma(pIMImaC,pIMBuffer3,IDFT);

    inFreeImaMem(pIMBuffer1);
    inFreeImaMem(pIMBuffer2);
    inFreeImaMem(pIMBuffer3);
  }
  else
  {
    inStatus = NK;
  }

  return(inStatus);
}

//-------------------------------------------------------------------------------------------------
-------------------------
//-------------------------------------------------------------------------------------------------
-------------------------
int inImaCorrelation(IMAGE *pIMImaC, IMAGE *pIMImaA, IMAGE *pIMImaB, int inCorType)
{
  int                           inStatus;
  IMAGE                     *pIMBuffer1,
                                   *pIMBuffer2;
```

```
  inStatus = OK;
  pIMBuffer1 = pIMAskImaMem(pIMImaA->inNumRows,pIMImaA->inNumCols);
  pIMBuffer2 = pIMAskImaMem(pIMImaA->inNumRows,pIMImaA->inNumCols);
  inHorFlipIma(pIMBuffer1,pIMImaA);
  inVerFlipIma(pIMBuffer2,pIMBuffer1);
  inFreeImaMem(pIMBuffer1);
  inImaConvolution(pIMImaC,pIMImaB,pIMBuffer2,inCorType);
  inFreeImaMem(pIMBuffer2);

  return(inStatus);
}

//-------------------------------------------------------------------------------------------------
-------------------------
//-------------------------------------------------------------------------------------------------
-------------------------
int inImaShift(IMAGE *pIMImaOut, IMAGE *pIMImaIn)
{
  int                           inStatus,
                                       inRowFirSeg,
                                       inRowSecSeg,
                                       inColFirSeg,
                                       inColSecSeg;
  IMAGE                    *pIMIma1;

  inStatus = OK;

  inRowFirSeg = ((int)ceil(((float)pIMImaIn->inNumRows)/2));
  inRowSecSeg = pIMImaIn->inNumRows-inRowFirSeg;
  inColFirSeg = ((int)ceil(((float)pIMImaIn->inNumCols)/2));
  inColSecSeg = pIMImaIn->inNumCols-inColFirSeg;

  pIMIma1 = pIMAskImaMem(inRowFirSeg,inColFirSeg);
  inExtractIma(pIMIma1,pIMImaIn,0,0,inRowFirSeg,inColFirSeg);
  inInsertIma(pIMImaOut,pIMIma1,pIMImaOut->inNumRows-inRowFirSeg,pIMImaOut->inNumCols-
inColFirSeg);
  inFreeImaMem(pIMIma1);

  pIMIma1 = pIMAskImaMem(inRowSecSeg,inColFirSeg);
  inExtractIma(pIMIma1,pIMImaIn,inRowFirSeg,0,inRowSecSeg,inColFirSeg);
  inInsertIma(pIMImaOut,pIMIma1,0,pIMImaOut->inNumCols-inColFirSeg);
  inFreeImaMem(pIMIma1);

  pIMIma1 = pIMAskImaMem(inRowFirSeg,inColSecSeg);
  inExtractIma(pIMIma1,pIMImaIn,0,inColFirSeg,inRowFirSeg,inColSecSeg);
  inInsertIma(pIMImaOut,pIMIma1,pIMImaOut->inNumRows-inRowFirSeg,0);
  inFreeImaMem(pIMIma1);

  pIMIma1 = pIMAskImaMem(inRowSecSeg,inColSecSeg);
  inExtractIma(pIMIma1,pIMImaIn,inRowFirSeg,inColFirSeg,inRowSecSeg,inColSecSeg);
  inInsertIma(pIMImaOut,pIMIma1,0,0);
  inFreeImaMem(pIMIma1);

  return(inStatus);
}

//-------------------------------------------------------------------------------------------------
-------------------------
//-------------------------------------------------------------------------------------------------
-------------------------
int inOTFFromAperture(IMAGE *pIMOTF, IMAGE *pIMAperture, int inConType)
{
  int                           inStatus,
                                       inCount1,
                                       inCount2;
  float                         flMax;
  mcomplex                      mcValue;
  IMAGE                    *pIMBuffer1,
                                   *pIMBuffer2;

  inStatus = OK;

  pIMBuffer1 = pIMAskImaMem(pIMAperture->inNumRows,pIMAperture->inNumCols);
  pIMBuffer2 = pIMAskImaMem(pIMAperture->inNumRows,pIMAperture->inNumCols);
```

```
  inCopyIma(pIMBuffer1,pIMAperture);

  inImaCorrelation(pIMBuffer2,pIMAperture,pIMBuffer1,inConType);
  inImaFunOp(pIMBuffer1,pIMBuffer2,IMRA);

  flMax = 0;
  for (inCount1=0;inCount1<pIMBuffer1->inNumRows;inCount1++)
  {
    for (inCount2=0;inCount2<pIMBuffer1->inNumCols;inCount2++)
       {
      inIMAGetImaElem(&mcValue,pIMBuffer1,inCount1,inCount2);
      if (flMax < mcValue.flRe)
        flMax = mcValue.flRe;
       }
  }
  voSetCom(&mcValue,flMax,0);
  inImaScaOp(pIMOTF,pIMBuffer1,mcValue,DIV);

  inFreeImaMem(pIMBuffer1);
  inFreeImaMem(pIMBuffer2);

  return(inStatus);
}

//-------------------------------------------------------------------------------------------------
--------------------------
//-------------------------------------------------------------------------------------------------
--------------------------
int inImaMetric(float *pflDistance, IMAGE *pIMA, IMAGE *pIMB, int inMetricType, float flParam)
{
  int                          inStatus,
                                      inCount1,
                                      inCount2;

  float                        flBuffer,
                                      doNorm;

  mcomplex                     mcMeanA,
                                      mcMeanB,
                                      mcVarA,
                                      mcVarB,
                                      mcBuffer1,
                                      mcBuffer2,
                                      mcCrossCorr;

  inStatus = OK;

  switch (inMetricType)
  {
    case LPNORM:
         flBuffer = 0;
      for (inCount1=0;inCount1<pIMA->inNumRows;inCount1++)
      {
        for (inCount2=0;inCount2<pIMA->inNumCols;inCount2++)
              {
         voSubCom(&mcBuffer1,pIMA->ppmcElem[inCount1][inCount2],pIMB-
>ppmcElem[inCount1][inCount2]);
         voMagCom(&doNorm,mcBuffer1);
            flBuffer += (float)pow(doNorm,flParam);
          }
      }
         (*pflDistance) = (float)pow(flBuffer/(pIMA->inNumRows*pIMA->inNumCols),1/flParam);
        break;

       case CORRELATION:
      inImaMean(&mcMeanA,pIMA);
      inImaMean(&mcMeanB,pIMB);

      inImaVar(&mcVarA,pIMA);
      inImaVar(&mcVarB,pIMB);

      voSetCom(&mcCrossCorr,0,0);
      for (inCount1=0;inCount1<pIMA->inNumRows;inCount1++)
         {
           for (inCount2=0;inCount2<pIMA->inNumCols;inCount2++)
              {
             voConjugateCom(&mcBuffer1,pIMB->ppmcElem[inCount1][inCount2]);
         voMulCom(&mcBuffer2,pIMA->ppmcElem[inCount1][inCount2],mcBuffer1);
```

```
            voAssignCom(&mcBuffer1,mcCrossCorr);
            voAddCom(&mcCrossCorr,mcBuffer1,mcBuffer2);
                }
        }

        voAssignCom(&mcBuffer1,mcCrossCorr);
        voSetCom(&mcBuffer2,(float)(pIMA->inNumRows*pIMA->inNumCols),0);
        voDivCom(&mcCrossCorr,mcBuffer1,mcBuffer2);

        voConjugateCom(&mcBuffer1,mcMeanB);
        voMulCom(&mcBuffer2,mcMeanA,mcBuffer1);
        voAssignCom(&mcBuffer1,mcCrossCorr);
        voSubCom(&mcCrossCorr,mcBuffer1,mcBuffer2);

        voMagCom(&flBuffer,mcCrossCorr);
            (*pflDistance) = flBuffer;
        voMagCom(&flBuffer,mcVarA);
            if (flBuffer == 0)
              (*pflDistance) = 0;
            else
              (*pflDistance) /= (float)sqrt(flBuffer);
        voMagCom(&flBuffer,mcVarB);
            if (flBuffer == 0)
              (*pflDistance) = 0;
            else
              (*pflDistance) /= (float)sqrt(flBuffer);

            break;

         default:
           inStatus = NK;

           break;
    }

    return(inStatus);
}

//-------------------------------------------------------------------------------------------------
--------------------------
//-------------------------------------------------------------------------------------------------
--------------------------
int inSimMap(IMAGE *pIMMap, IMAGE *pIMRef, IMAGE *pIMIma, int inBlockSize, int inMetricType,
float flParam)
{
    int                         inStatus,
                                        inCount1,
                                        inCount2,
                                        inCount3,
                                        inCount4;
    float                       flValue;
    IMAGE                   *pIMBuffer1,
                                    *pIMBuffer2;

    inStatus = OK;

    pIMBuffer1 = pIMIMAAskImaMem(inBlockSize,inBlockSize);
    pIMBuffer2 = pIMIMAAskImaMem(inBlockSize,inBlockSize);

    for (inCount1=0;inCount1<pIMRef->inNumRows;inCount1+=inBlockSize)
    {
      for (inCount2=0;inCount2<pIMRef->inNumCols;inCount2+=inBlockSize)
          {
        inExtractIma(pIMBuffer1,pIMRef,inCount1,inCount2,inBlockSize,inBlockSize);
        inExtractIma(pIMBuffer2,pIMIma,inCount1,inCount2,inBlockSize,inBlockSize);

        inImaMetric(&flValue,pIMBuffer1,pIMBuffer2,inMetricType,flParam);

            for (inCount3=inCount1;inCount3<inCount1+inBlockSize;inCount3++)
            {
          for (inCount4=inCount2;inCount4<inCount2+inBlockSize;inCount4++)
                {
            voSetCom(&(pIMMap->ppmcElem[inCount3][inCount4]),flValue,0);
                }
            }
        }
```

```
  }

  inIMAFreeImaMem(pIMBuffer1);
  inIMAFreeImaMem(pIMBuffer2);

  return(inStatus);
}

//-------------------------------------------------------------------------------------------------------
-------------------------
//-------------------------------------------------------------------------------------------------------
-------------------------
int inGetWindow(IMAGE *pIMWindow, int inWindowType)
{
  int                           inStatus,
                                       inCount1,
                                       inCount2;
  float                         flRowCen,
                                       flColCen,
                                       flCenRow,
                                       flCenCol,
                                       flExtRow,
                                       flExtCol,
                                       flParLength,
                                       flTotLength,
                                       flPos;

  inStatus = OK;
// CACA
  flRowCen = (float)(pIMWindow->inNumRows/2);
  flColCen = (float)(pIMWindow->inNumCols/2);

  for (inCount1=0;inCount1<pIMWindow->inNumRows;inCount1++)
  {
    for (inCount2=0;inCount2<pIMWindow->inNumCols;inCount2++)
       {
      flCenCol = ((float)inCount2)-flColCen;
      flCenRow = ((float)inCount1)-flRowCen;

         if ((flCenCol == 0) && (flCenRow == 0))
         {
        flPos = 0.5;
         }
         else
         {
           if (flCenCol == 0)
              {
          flExtCol = 0;
          flExtRow = flRowCen;
              }
           else
              {
          flExtCol = flColCen;
             flExtRow = (float)fabs((flCenRow/flCenCol)*flColCen);

             if (flExtRow > flRowCen)
                {
                  flExtCol = (float)fabs((flCenCol/flCenRow)*flRowCen);
               flExtRow = flRowCen;
                }
              }

        flParLength = (float)sqrt(flCenCol*flCenCol+flCenRow*flCenRow);
           flTotLength = (float)(2*sqrt(flExtCol*flExtCol+flExtRow*flExtRow));

        flPos = (flTotLength/2+flParLength)/flTotLength;
         }

      switch (inWindowType)
         {
        case RECTANGULAR:
          voSetCom(&(pIMWindow->ppmcElem[inCount1][inCount2]),1,0);
          break;

        case HAMMING:
```

```
          voSetCom(&(pIMWindow->ppmcElem[inCount1][inCount2]),(float)(0.54-
0.46*cos(2*PI*flPos)),0);
          break;

        case HANNING:
          voSetCom(&(pIMWindow->ppmcElem[inCount1][inCount2]),(float)(0.50-
0.50*cos(2*PI*flPos)),0);
          break;

        case BLACKMAN:
          voSetCom(&(pIMWindow->ppmcElem[inCount1][inCount2]),(float)(0.42-
0.50*cos(2*PI*flPos)+0.08*cos(4*PI*flPos)),0);
          break;

        default:
          inStatus = NK;
             break;
         }
        }
  }

  return(inStatus);
}

//-------------------------------------------------------------------------------------------------
--------------------------
//-------------------------------------------------------------------------------------------------
--------------------------
int inLikelihood(float *pflLik, IMAGE *pIMg, IMAGE *pIMEstBlur, IMAGE *pIMf,
                                IMAGE *pIMPrior, int inType)

{
  int                           inStatus,
                                       inCount1,
                                       inCount2;
  float                         flBuffer1,
                                       flg,
                                       fleb,
                                       flf,
                                       flp;


  inStatus = OK;

  switch (inType)
  {
    case ML:
          flBuffer1 = 0;
      for (inCount1=0;inCount1<pIMg->inNumRows;inCount1++)
         {
        for (inCount2=0;inCount2<pIMg->inNumCols;inCount2++)
              {
          flg = pIMg->ppmcElem[inCount1][inCount2].flRe+1;
          fleb = pIMEstBlur->ppmcElem[inCount1][inCount2].flRe+1;

          flBuffer1 += (flg*((float)log(fleb))-fleb-flLogNatFac(flg));
              }
         }
      (*pflLik) = flBuffer1;
         (*pflLik) /= (pIMg->inNumRows*pIMg->inNumCols);
         break;

    case MAP:
          flBuffer1 = 0;
      for (inCount1=0;inCount1<pIMg->inNumRows;inCount1++)
         {
        for (inCount2=0;inCount2<pIMg->inNumCols;inCount2++)
              {
                flg = pIMg->ppmcElem[inCount1][inCount2].flRe+1;
                fleb = pIMEstBlur->ppmcElem[inCount1][inCount2].flRe+1;

          flBuffer1 += (flg*((float)log(fleb))-fleb-flLogNatFac(flg));
              }
         }
      (*pflLik) = flBuffer1;
          flBuffer1 = 0;
      for (inCount1=0;inCount1<pIMf->inNumRows;inCount1++)
         {
```

```
        for (inCount2=0;inCount2<pIMf->inNumCols;inCount2++)
               {
                 flf = pIMg->ppmcElem[inCount1][inCount2].flRe+1;
                 flp = pIMEstBlur->ppmcElem[inCount1][inCount2].flRe+1;

          flBuffer1 += (flf*((float)log(flp))-flp-flLogNatFac(flf));
               }
          }
      (*pflLik) += flBuffer1;
         (*pflLik) /= (pIMg->inNumRows*pIMg->inNumCols);
         break;

    default:
      inStatus = NK;
      (*pflLik) = 0;
         break;
  }

  return(inStatus);
}

//-------------------------------------------------------------------------------------------------------------
//-------------------------------------------------------------------------------------------------------------
float flLogNatFac(float flNumber)
{
  float                         flCoefficients[6];
  float                         flBuffer1,
                                        flBuffer2,
                                        flBuffer3,
                                        flBuffer4;

  flCoefficients[0] = (float)(+76.180091729471460000);
  flCoefficients[1] = (float)(-86.505320329416770000);
  flCoefficients[2] = (float)(+24.014098240830910000);
  flCoefficients[3] = (float)(-01.231739572450155000);
  flCoefficients[4] = (float)(+00.001208650973866179);
  flCoefficients[5] = (float)(-00.000005395239384953);

  flBuffer1 = flNumber+1;
  flBuffer2 = flBuffer1;
  flBuffer3 = (float)(flBuffer1+5.5-(flBuffer1+0.5)*log(flBuffer1+5.5));
  flBuffer4 = (float)(1.000000000190015);
  flBuffer4 += flCoefficients[0]/++flBuffer2;
  flBuffer4 += flCoefficients[1]/++flBuffer2;
  flBuffer4 += flCoefficients[2]/++flBuffer2;
  flBuffer4 += flCoefficients[3]/++flBuffer2;
  flBuffer4 += flCoefficients[4]/++flBuffer2;
  flBuffer4 += flCoefficients[5]/++flBuffer2;

  return(((float)log(2.5066282746310005*flBuffer4/flBuffer1))-flBuffer3);
}

//-------------------------------------------------------------------------------------------------------------
//-------------------------------------------------------------------------------------------------------------
int inTotalFlip(IMAGE *pIMOutIma, IMAGE *pIMInIma)
{
  int                           inStatus,
                                        inNumRow,
                                        inNumCol;
  IMAGE                      *pIMBuffer1,
                                     *pIMBuffer2;

  inStatus = OK;

  inNumRow = pIMInIma->inNumRows;
  inNumCol = pIMInIma->inNumCols;

  pIMBuffer1 = pIMIMAAskImaMem(inNumRow,inNumCol);
  pIMBuffer2 = pIMIMAAskImaMem(inNumRow,inNumCol);

  inInsertIma(pIMOutIma,pIMInIma,0,0);
```

```
    inHorFlipIma(pIMBuffer1,pIMInIma);
    inInsertIma(pIMOutIma,pIMBuffer1,inNumRow,0);

    inVerFlipIma(pIMBuffer1,pIMInIma);
    inInsertIma(pIMOutIma,pIMBuffer1,0,inNumCol);

    inHorFlipIma(pIMBuffer1,pIMInIma);
    inVerFlipIma(pIMBuffer2,pIMBuffer1);
    inInsertIma(pIMOutIma,pIMBuffer2,inNumRow,inNumCol);

    inFreeImaMem(pIMBuffer1);
    inFreeImaMem(pIMBuffer2);

    return(inStatus);
}
```

```
NAME
    mcomplex.h
DESCRIPTION
    Header file for mcomplex.cpp
PROGRAMMER
    Pablo Zegers
```

```
//---------------------------------------------------------------------------
#ifndef Fmcomplex
//---------------------------------------------------------------------------

//---------------------------------------------------------------------------
// INCLUDE FILES
//---------------------------------------------------------------------------
#include "..\Global\Global.h"

//---------------------------------------------------------------------------
// STRUCTURE DEFINITIONS
//---------------------------------------------------------------------------
typedef struct
{
  float                 flRe,
                                    flIm;
} mcomplex;

//---------------------------------------------------------------------------
// PUBLIC PROTOTYPES
//---------------------------------------------------------------------------
void voSetCom(mcomplex *pmcA, float flRe, float flIm);
void voAssignCom(mcomplex *pmcB, mcomplex mcA);
void voConjugateCom(mcomplex *pmcB, mcomplex mcA);
void voMagCom(float *pflMag, mcomplex mcA);
void voPhaCom(float *pflPhase, mcomplex mcA);
void voAddCom(mcomplex *pmcC, mcomplex mcA, mcomplex mcB);
void voSubCom(mcomplex *pmcC, mcomplex mcA, mcomplex mcB);
void voMulCom(mcomplex *pmcC, mcomplex mcA, mcomplex mcB);
void voDivCom(mcomplex *pmcC, mcomplex mcA, mcomplex mcB);
void voExpCom(mcomplex *pmcB, mcomplex mcA);
void voLogCom(mcomplex *pmcB, mcomplex mcA);
void voPowCom(mcomplex *pmcB, mcomplex mcA, mcomplex mcB);

//---------------------------------------------------------------------------
#define Fmcomplex             0
#endif
//---------------------------------------------------------------------------
```

NAME
mcomplex.cpp
DESCRIPTION
Routines neede to manipulate complex numbers.
PROGRAMMER
Pablo Zegers

```
//-----------------------------------------------------------------------------
// DESCRIPTION : Functions to manipulate complex numbers
// AUTHOR      : Pablo Zegers
// DATE        : March, 1998
//-----------------------------------------------------------------------------

//-----------------------------------------------------------------------------
// INCLUDE FILES
//-----------------------------------------------------------------------------
#include <stdlib.h>
#include <stdio.h>
#include <io.h>
#include <string.h>
#include <malloc.h>
#include <math.h>
#include "mcomplex.h"

//-----------------------------------------------------------------------------
// PUBLIC ROUTINES
//-----------------------------------------------------------------------------

//-----------------------------------------------------------------------------
//-----------------------------------------------------------------------------
void voSetCom(mcomplex *pmcA, float flRe, float flIm)
{
  pmcA->flRe = flRe;
  pmcA->flIm = flIm;
}

//-----------------------------------------------------------------------------
//-----------------------------------------------------------------------------
void voAssignCom(mcomplex *pmcB, mcomplex mcA)
{
  pmcB->flRe = mcA.flRe;
  pmcB->flIm = mcA.flIm;
}

//-----------------------------------------------------------------------------
//-----------------------------------------------------------------------------
void voConjugateCom(mcomplex *pmcB, mcomplex mcA)
{
  pmcB->flRe = mcA.flRe;
  pmcB->flIm = -mcA.flIm;
}

//-----------------------------------------------------------------------------
//-----------------------------------------------------------------------------
void voMagCom(float *pflMag, mcomplex mcA)
{
  (*pflMag) = (float)sqrt(mcA.flRe*mcA.flRe+mcA.flIm*mcA.flIm);
}

//-----------------------------------------------------------------------------
//-----------------------------------------------------------------------------
void voPhaCom(float *pflPhase, mcomplex mcA)
{
  (*pflPhase) = (float)atan(mcA.flIm/mcA.flRe);
}

//-----------------------------------------------------------------------------
//-----------------------------------------------------------------------------
void voAddCom(mcomplex *pmcC, mcomplex mcA, mcomplex mcB)
{
  pmcC->flRe = mcA.flRe+mcB.flRe;
  pmcC->flIm = mcA.flIm+mcB.flIm;
}

//-----------------------------------------------------------------------------
```

```
//
//-----------------------------------------------------------------------------
void voSubCom(mcomplex *pmcC, mcomplex mcA, mcomplex mcB)
{
  pmcC->flRe = mcA.flRe-mcB.flRe;
  pmcC->flIm = mcA.flIm-mcB.flIm;
}

//-----------------------------------------------------------------------------
//-----------------------------------------------------------------------------
void voMulCom(mcomplex *pmcC, mcomplex mcA, mcomplex mcB)
{
  pmcC->flRe = mcA.flRe*mcB.flRe-mcA.flIm*mcB.flIm;
  pmcC->flIm = mcA.flRe*mcB.flIm+mcA.flIm*mcB.flRe;
}

//-----------------------------------------------------------------------------
//-----------------------------------------------------------------------------
void voDivCom(mcomplex *pmcC, mcomplex mcA, mcomplex mcB)
{
  float                         flMag2;

  pmcC->flRe = mcA.flRe*mcB.flRe+mcA.flIm*mcB.flIm;
  pmcC->flIm = mcA.flIm*mcB.flRe-mcA.flRe*mcB.flIm;

  flMag2 = mcB.flRe*mcB.flRe+mcB.flIm*mcB.flIm;

  if (flMag2 != 0)
  {
    pmcC->flRe /= flMag2;
    pmcC->flIm /= flMag2;
  }
}

//-----------------------------------------------------------------------------
//-----------------------------------------------------------------------------
void voExpCom(mcomplex *pmcB, mcomplex mcA)
{
  pmcB->flRe = (float)(exp(mcA.flRe)*cos(mcA.flIm));
  pmcB->flIm = (float)(exp(mcA.flRe)*sin(mcA.flIm));
}

//-----------------------------------------------------------------------------
//-----------------------------------------------------------------------------
void voLogCom(mcomplex *pmcB, mcomplex mcA)
{
  float                                 flMag,
                                               flPhase;

  flMag = (float)sqrt(mcA.flRe*mcA.flRe+mcA.flIm*mcA.flIm);
  if (mcA.flRe != 0)
  {
    flPhase = (float)atan(mcA.flIm/mcA.flRe);
  }
  else
  {
    if (mcA.flIm > 0)
      flPhase = (float)(+PI/2);
       else
      flPhase = (float)(-PI/2);
  }

  pmcB->flRe = (float)(log(flMag));
  pmcB->flIm = flPhase;
}

//-----------------------------------------------------------------------------
//-----------------------------------------------------------------------------
void voPowCom(mcomplex *pmcB, mcomplex mcA, mcomplex mcB)
{
  mcomplex                              mcBuffer1,
                                               mcBuffer2;

  voLogCom(&mcBuffer1,mcA);
  voMulCom(&mcBuffer2,mcB,mcBuffer1);
  voExpCom(pmcB,mcBuffer2);
```